Asynchronous sessions with implicit functions and messages

Article (Accepted Version)

Asynchronous Sessions with Implicit Functions and Messages[☆]

Alex Jeffery[a,*], Martin Berger[a]

[a] *University of Sussex, Falmer, Brighton, BN1 9RH, United Kingdom*

Abstract

Session types are a well-established approach to ensuring protocol conformance and the absence of communication errors such as deadlocks in message passing systems. Haskell introduced implicit parameters, Scala popularised this feature and recently gave implicit types first-class status, yielding an expressive tool for handling context dependencies in a type-safe yet terse way. We ask: can type-safe implicit functions be generalised from Scala's sequential setting to message passing computation? We answer this question in the affirmative by generalising the concept of an implicit function to an *implicit message*, its concurrent analogue. We present two calculi, each with implicit message passing. The first, Im, is a concurrent functional language that extends Gay and Vasconcelos's calculus of linear types for asynchronous sessions (Last) with implicit functions and messages. The second, MpIm, is a π-calculus with implicit messages that extends Coppo, Dezani-Ciancaglini, Padovani and Yoshida's calculus of multiparty asynchronous sessions (Mpst). We argue, via examples, that these new language features provide utility to the programmer, and prove each system sound by translation into its respective base calculus.

Keywords: implicits, session types, asynchronous session types, multiparty session types, Scala, type classes, Haskell, concurrency, type system

1. Introduction

1.1. Session types

Types classify programs, distinguishing between programs that are guaranteed to exhibit semantic properties of interest, and those that are not. Types

[☆] Full version of [9] with proofs, and a new section on implicits for multi-party asynchronous sessions.

Abbreviations: Last, Lambda calculus with Asynchronous Session Types; Im, the calculus of Implicit functions and Messages; Mpst, the calculus of MultiParty Session Types; MpIm, the calculus of MultiParty session types and Implicit Messages.

[*] Corresponding author

Email addresses: A.P.Jeffery@sussex.ac.uk (Alex Jeffery), M.F.Berger@sussex.ac.uk (Martin Berger)

for sequential computation are well-established and a core part of industrial software engineering. Behavioural type systems extend types to concurrent, parallel and distributed computation, and are a core activity of contemporary type theoretical research.

Session types, first introduced in [8, 19], are an important example of a behavioural type system for message passing concurrency. Session types classify message passing behaviour at given channels: e.g. if process P first sends an integer on channel x, then receives a boolean on x, and finally sends a boolean on x, then this behaviour could be summarised by the session type

P : `!Int.?Bool.!Bool.end`

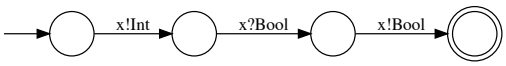
Here `?T` represents input of a value of type `T`, `!T` means sending a value that has type `T`, while `end` denotes the termination of the interaction.

A key notion in session types is that of *duality*, originating in linear logic: processes P and Q can be composed in parallel only when throughout the course of the computation each output of P's is matched by a suitable input of Q's, and vice versa. In this case we say that P and Q are *dual*. Session types ensure that only dual processes are composed in parallel. Hence typability guarantees the absence of communication errors such as mismatched communication and deadlocks. A process Q, dual to P above, would have the session type

Q : `?Int.!Bool.?Bool.end`

Notice that for each action in P's type, we have the dual action in Q's type, e.g. an output of type `!Int` can be received by an input of type `?Int`.

Session types can be viewed as finite state automata, with edges classifying message send/receive actions, and constraining causality between message exchange. For example, the behaviour of the process P above corresponds to the following FSA:



Unlike traditional automata, session types are built up compositionally from program syntax.

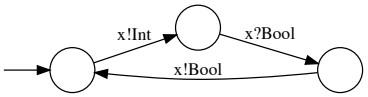
Recursive types, a key component of session types, allow for the description of protocols containing looping and repetition. Consider P', a variant of P above, that now instead of just sending an integer, receiving a boolean and sending a boolean, now performs those same three actions repeatedly. We denote such a session type:

P' : μ`X.!Int.?Bool.!Bool.X`

or, alternatively:

P' : `let X = !Int.?Bool.!Bool.X in X`

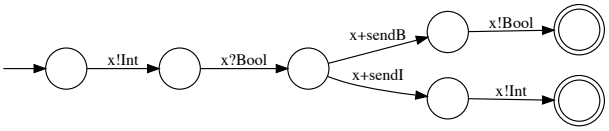
A corresponding finite state automaton for P' is shown below:



Further important components of session types are the dual notions of internal and external choice. Consider now our previous example P, further augmented thusly: instead of sending a boolean as the final action, the process now makes an *external choice*, where a selection is offered of either (1) sending a boolean as before, or (2) sending an integer. The dual process may select either option in an *internal choice*. We express this new behaviour with the session type P'':

P'' : `!Int.?Bool.&⟨sendB:!Bool.end, sendI:!Int.end⟩`

The tag `sendB` represents the boolean option, whereas `sendI` represents the integer option. This session type can be visualised by the following automaton:

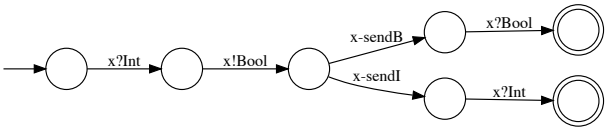


This external choice type represents a selection of services offered by a server P''. The dual process Q'' is therefore allowed to make an internal choice, whereby it selects from the services on offer. The process P'' must have a matching external choice of type S for every internal choice of type T that Q can make, such that S and T are dual. Note that the converse does not always hold - while it is clear that we must never allow Q'' to select a service that P'' does not offer, we might allow P'' to offer a choice that Q'' never selects.

In the above case, we obtain the following dual type Q'':

Q'' : `?Int.!Bool.⊕⟨sendB:?Bool.end, sentI:?Int.end⟩`

This dual type corresponds to the following automaton:



1.2. Last

Gay and Vasconcelos's calculus Last (Lambda calculus with Asynchronous Session Types) [7] is a concurrent functional programming language, the first coherent integration of session types with λ-calculus. Processes (essentially functional programs) can be composed in parallel along with message buffers. Processes send messages that are placed in the message buffers, from where they are later asynchronously retrieved by other processes. Channels are held by processes and are used to specify which buffers receive which messages. Binary session types are imposed on the channels to ensure that communication patterns between these processes are always dual. Message exchange is achieved via `send` and `receive` combinators, which have the following types:

```
send :: T → !T.S → S
receive :: ?T.S → T ⊗ S
```

These types follow standard intuition about the behaviour of the respective constructs. The `send` combinator takes two arguments - a message of type `T`, and a channel which expects to perform communication of type `!T.S`. The `send` combinator delivers the message of type `T` along the channel, which we see from the channels type that it expects. The value returned is a channel of type `S` - the behaviour that we expect from the channel after the send operation. The `receive` combinator takes just one argument of type `?T.S` - the channel on which the process expects to receive a message - and returns a pair `T ⊗ S` - the received message and a channel on which communication can continue.

In Last, session types are imposed via linearity constraints on channel names: each channel is used exactly once, and the interaction subsequently continues on a channel returned by the previous interaction. If linear channel usage was not enforced, processes could violate their session types in a number of ways, for example, by attempting to receive a single message twice, which would cause the receiver to wait indefinitely, or by neglecting to send a message required by the session type.

The usage pattern of performing an interaction on a channel, and then rebinding the returned channel for the next interaction is forced upon us by linear typing of channels. This results in Last programs containing many `let` constructs, since every interaction requires the binding of a new channel. This creates syntactic noise in programs, which is arguably undesirable. In section 2.1, we show how implicit functions can provide a solution to this problem.

Shown below is an example Last program, in which two communication partners initiate a session (via the `accept` and `request` operations). Each communication partner is a process, which consists of an expression enclosed in $\langle$ angular brackets $\rangle$. These processes are separated by the *parallel composition* operator $\|$, which indicates that the processes run concurrently, and can communicate if they each have one of the two dual endpoints of a communication channel. One process sends the integer to the other, and the other replies with the incremented received integer.

```
⟨ let    c = request x      in
  let    c = send 10      c in
  let m, c = receive      c in m    ⟩ ‖
⟨ let    d = accept x       in
  let n, d = receive      d in
  let    d = send (n + 1) d in unit ⟩
```

The session type for the first process is `?Int.!Int.end` and the type for the second is `!Int.?Int.end`. The first reduction is the session initiation, which creates two buffers (one for each direction) that the processes use for communication. The dual endpoints `c` and `d` to a shared communication channel are created, which can be used by each process to access these buffers. In general, a buffer $\mathtt{a} \to (\mathtt{b}, q)$ stores messages in a queue q sent on the channel `b`, that are to be received on the channel `a`. Messages are appended to the right of q when sent, and removed from the left when received. Session initiation also creates a restriction over the names `c` and `d` to prevent interference in the session by other processes. The session initiation yields:

```
(νcd)(
⟨ let    c = send 10      c in
  let m, c = receive      c in m    ⟩ ‖
⟨ let n, d = receive      d in
  let    d = send (n + 1) d in unit ⟩ ‖
c → (d, ϵ)  ‖  d → (c, ϵ))
```

Subsequently the second process performs the operation `send 10 c`, placing the value `10` in the buffer for the channel `d`:

```
(νcd)(
⟨ let m, c = receive      c in m    ⟩ ‖
⟨ let n, d = receive      d in
  let    d = send (n + 1) d in unit ⟩ ‖
c → (d, ϵ)  ‖  d → (c, 10))
```

In the next reduction step, the second process retrieves the value `10` from its buffer, and it is substituted for `n`:

```
(νcd)(
⟨ let m, c = receive       c in m    ⟩ ‖
⟨ let    d = send (10 + 1) d in unit ⟩ ‖
c → (d, ϵ)  ‖  d → (c, ϵ))
```

At this point, the subexpression `(10 + 1)` reduces to `11` (we omit this step for brevity). Following this, the first process deposits the result in the buffer for channel `c`:

```
(νcd)(
⟨ let m, c = receive c in m ⟩ ‖
⟨ unit                      ⟩ ‖
c → (d, 11)  ‖  d → (c, ϵ))
```

Finally the second process retrieves the value `11` from the buffer.

```
(νcd)(⟨ 11 ⟩ ‖ ⟨ unit ⟩ ‖ c → (d, ϵ) ‖ d → (c, ϵ))
```

1.3. Multiparty session types

The session types we have seen thus far have been *binary* session types, in which there are exactly two participants whose actions are dual to one another. It is possible to generalise this binary form of session to an arbitrary number of participants, a so-called *multiparty* session. Consider the following protocol with three participants, with *participant numbers* `1`, `2` and `3`:

- First, participant `1` sends an integer to participant `2`.
- Then, participant `2` sends a boolean to participant `3`.
- Finally, participant `3` sends a string to participant `1`.

Instead of looking at this protocol from two dual points of view, we describe it with a type that takes a *global* view of all communication. We describe the above protocol with the following *global type* G:

```
1→2:⟨Int⟩.2→3:⟨Bool⟩.3→1:⟨String⟩.end
```

In the above, our communication types are of the form $\mathtt{p} \to \mathtt{q} : \langle T \rangle$, which tells us a message is sent by `p`, received by `q`, and that the message content is of type T.

It is possible to view this protocol from the point of view of any of the three participants, or in other words, to *project* the global type G onto a participant `p` ($1 \leq \mathtt{p} \leq 3$), obtaining a *local* session type. We denote this projection $G \upharpoonright \mathtt{p}$. In this case, $G \upharpoonright \mathtt{1}$ yields:

```
!⟨2,Int⟩.?⟨3,String⟩.end
```

$G \upharpoonright \mathtt{2}$ yields:

```
?⟨1,Int⟩.!⟨3,Bool⟩.end
```

$G \upharpoonright \mathtt{3}$ yields:

```
?⟨2,Bool⟩.!⟨1,String⟩.end
```

In the local view we need only write a single participant number, since the participation of the local participant is implied. We write ! or ? to indicate whether the local participant does the sending or the receiving.

We can further project local session types onto another participant `q`, to obtain a type that describes only the communication between `p` and `q`. The syntax of such types matches the syntax of binary session types. Projecting a second participant number onto the above protocol leads to the following types:

$$
\begin{array}{llll}
G \upharpoonright \mathtt{1} \upharpoonright \mathtt{2} & = & \mathtt{!Int.end} & \qquad G \upharpoonright \mathtt{1} \upharpoonright \mathtt{3} = \mathtt{?String.end} \\
G \upharpoonright \mathtt{2} \upharpoonright \mathtt{1} & = & \mathtt{?Int.end} & \qquad G \upharpoonright \mathtt{3} \upharpoonright \mathtt{1} = \mathtt{!String.end} \\
G \upharpoonright \mathtt{2} \upharpoonright \mathtt{3} & = & \mathtt{!Bool.end} & \qquad G \upharpoonright \mathtt{3} \upharpoonright \mathtt{2} = \mathtt{?Bool.end}
\end{array}
$$

Now that we have recovered binary session types by two projections, observe that for all participants `p`, `q` in G, $G \upharpoonright \mathtt{p} \upharpoonright \mathtt{q}$ is dual to $G \upharpoonright \mathtt{q} \upharpoonright \mathtt{p}$. This property holds for the above example, and is a condition of typability for all multiparty session-typed programs.

1.4. Implicit functions

Modularity, a core concept in software engineering, is greatly aided by parameterisation of programs. Parameterisation has dual facets: supplying and consuming a parameter. A key tension in large-scale software engineering is between *explicit* (e.g. pure functional programming), and *implicit* parameterisation (e.g. global state). The former enables local reasoning but can lead to repetitive supply of parameters. Here is a simple example of the problem (where <= is the function $\lambda xy.x \leq y$, and α a type):

```
let f x compare : α = ... in
    f 3 (<=)
    ...
    f 17 (<=)
    ...
```

Repeatedly passing functions like <= which are unlikely to change frequently, is tedious, and impedes readability of large code bases. Default parameters are an early proposal for mediating this tension in a type-safe way. The key idea is to annotate function arguments with their default value, to be used whenever an invocation does not supply an argument:

```
let f x (compare = (<=)) : α = ... in
    f 2
    ...
    f 5
    ...
```

The compiler synthesises `f 2 (<=)` from `f 2`, and `f 5 (<=)` from `f 5`. Default parameters have a key disadvantage: the default value is hard-coded at the callee, and cannot be context dependent. Implicit arguments, a strict generalisation of default parameters, were pioneered in Haskell [12], and popularised as well as refined in Scala [14]: they separate the *callee's declaration* that an argument can be elided, from the *caller's choice* of elided values, allowing the latter to be context dependent.

```
let f x (implicit compare) : α = ... in
    let implicit cmp = (<=) in
        f 2
    ...
    let implicit cmp = (>)  in
        f 5
    ...
```

In this example `f 2` is rewritten as above, but `f 5` becomes `f 5 (>)`, i.e. a different implicit argument is synthesised. The disambiguation between several providers of implicit arguments happens at compile-time using type and scope information. Programs where elided arguments cannot be disambiguated at compile-time are rejected as ill-formed. Hence type-safety is not compromised.

One might ask: can type-safe implicit functions be generalised from Scala's sequential setting to message passing computation? We answer this question in the affirmative by generalising the concept of implicit functions to *implicit messages*. We elaborate on this idea by presenting the first concurrent func-

tional language with implicit message passing, called Im. Im is an extension to Last that adds implicit message passing and implicit functions. We use several examples to argue that implicit messages provide useful abstractions for programming languages with session types. Additionally, we show how implicit functions can be used to remove repeated rebinding of channel names. Im is presented in detail in sections 2 through 6.

1.5. Implicit messages

Generalising from the sequential case, the concept of implicit messages has two dual parts:

- Input can be declared implicit, and need not be explicitly matched by an output in the dual process.
- At compile time, suitable output is synthesised, based on type and scope information.

In the following example, we have two parallel processes `p` and `q` . They initiate a session, whereafter `p` performs an (implicit) receive and `q` apparently does nothing.

```
⟨ let p =
    let c = accept x in
    let n, c = implicit receive c in c
in p ⟩ ‖ ⟨ let q =
    let ⸮ = 10 in
    let d = request x in d
in q ⟩
```

The type system sees the implicit receive in `p` and is able to figure out that a corresponding send must be inserted into `q`. It knows that the channel that the send occurs on is `d` since it is the dual channel to `c` which the `implicit receive` uses. The chosen message is a variable of appropriate type from the implicit scope - the construct `let ⸮ = ...` introduces an implicit variable to the implicit scope. The implicit scope can be thought of as a store of variables that are designated as implicit - we make this notion more precise in Section 5. Following [14], we do not give names to implicit variables until after translation, but use ⸮ (pronounced 'query') as a placeholder name for all implicit variables. The translation becomes:

```
⟨ let p =
    let c = accept x in
    let n, c = receive c in c
in p ⟩ ‖ ⟨ let q =
    let y = 10 in
    let d = request x in d
    send y d
in q ⟩
```

Here `y` is a fresh variable. The insertion of this fresh variable `y` and the synthesised output `send y d` correspond to adding an implicit variable as an additional argument to an implicit function in Scala.

One might ask: are implicit messages limited to binary communication only, or can they be made to work in a multiparty scenario? We show that they can indeed work in a multipary scenario by adding implicit messages to a π-calculus with multiparty session types. We term the resulting language MpIm. MpIm is introduced in sections 7 through 10.

2. Im - Further examples

2.1. Elimination of repeated rebinding

A well-known problem with the integration of session types and sequential languages is the seeming necessity of repeated rebinding of channel names. The problem is that `send` takes a channel of type $!T.S$ as its second argument, and returns a linear channel of type S. In order for linearity to be respected that channel must be rebound. Consider the process below, typical of LAST programs. Note that the `select` combinator corresponds to the type $\oplus\langle ...\texttt{l:S}, ...\rangle$ introduced in section 1.1, and the process below selects from the paths `label1` or `label2` offered by its dual, based on the result of `pred(m)`.

```
miscService :: ⟨S⟩^a → end
miscService ap =
    let    c = accept ap in
    let m, c = receive c in
    let n, c = receive c in
    if pred(m) then
        let c = select label1 c in
        let c = send f(m, n) c in
        let c = send g(m, n, n) c in c
    else
        let c = select label2 c in
        let o, c = receive c in
        let c = send f(n, m) c in
        let c = send g(m, n, o) c in c
```

This redundancy makes programs hard to read. The issue can be addressed in other ways, for example using parameterised monads [3], see also [7, Chapter 7]. Implicit functions and message passing enable a principled and canonical two-step solution: (1) make the channel argument implicit and let the compiler synthesise the missing channel name for rebinding; (2) include in the language two special constructs: `let`$^\wr$, which unpacks a pair of form *(value, channel)*, such that the left hand value is bound to a given name, and the right hand channel is bound to the implicit variable; and `;`$^\wr$, which binds a single channel variable, resultant on the left, to the implicit scope of the computation on the right. This solution is sufficient for languages that are not known to admit monads, and requires only implicit functions.

The `send` primitive has type $T \to !T.S \to S$. We can use implicit function types to define a new output primitive `send`$^\wr$, with type $T \to !T.S \wr\!\!\to S$, explained below. The annotation $\wr$ in $!T.S \wr\!\!\to S$ makes the channel argument implicit - the message will be sent on a channel in the implicit scope with the appropriate session type.

```
send≀ :: T → !T.S ↣ S
send≀ m = send m ≀
```

We can do something similar for `select` and `receive`.

```
select≀ :: Label l → ⊕⟨...l:S,...⟩ ↣ S
select≀ l = select l ≀

receive≀ :: ?T.S ↣ T ⊗ S
receive≀ = receive ≀
```

We define `let`$^{\wr}$ and `;`$^{\wr}$ as follows:

$$\begin{aligned} \texttt{let}^{\wr}\ x = e_1\ \texttt{in}\ e_2 &\stackrel{\text{def}}{=} \texttt{let}\ x, \wr = e_1\ \texttt{in}\ e_2 \\ e_1\ \texttt{;}^{\wr}\ e_2 &\stackrel{\text{def}}{=} \texttt{let}\ \wr = e_1\ \texttt{in}\ e_2 \end{aligned}$$

We can rewrite `miscService` above with our new primitives. The resulting code is less repetitive and more terse, hence readable.

```
miscService :: ⟨S⟩ᵃ ↣ end
miscService =
    let ≀ = accept ≀ in
    let m = receive≀ in
    let n = receive≀ in
    if pred(m) then
        select≀ label1 ;≀
        send≀ f(m, n) ;≀
        send≀ g(m, n, n)
    else
        select≀ label2 ;≀
        let o = receive≀ in
        send≀ f(n, m) ;≀
        send≀ g(m, n, o)
```

We believe that it is possible to omit the superscript ≀ annotations, using type inference to distinguish ordinary `let` constructs from our new `let`$^{\wr}$ constructs. `;`$^{\wr}$ constructs could likely be omitted and programs appropriately augmented in a similar way. An implementation could conceivably try to type check the program with normal `let` and `;` constructs, and replace them with implicit ones if they cause the type check to fail. Scala's implicit conversions are implemented with such a heuristic [18], and such a technique is likely applicable here.

Our solution to the rebinding problem is robust, and applicable to linear types generally. [5] presents a linear typing system for Haskell, and demonstrates the use of linear functions in Haskell. A prototypical use case of linear functions in Haskell is to prevent common file errors, such as writing to a closed file or double closure. Linear Haskell provides a set of functions that take as a linear parameter a file, which is then returned and rebound. Some examples of such functions and their types are given next:

```
openFile :: FilePath → IO_L 1 File
readLine :: File ⊸ IO_L 1 (File, Unrestricted Bytestring)
closeFile :: File ⊸ IO_L ω ()
```

Here IO_L is a type constructor that represents types obtained by doing IO, and the *multiplicities* 1 or ω denote whether or not the type is linear - 1 for linear, ω for unrestricted. A typical usage example of these functions might be:

```
do f <- openFile "myFile.txt"
   line, f <- readLine f
   if somePredicate line then
       line2, f <- readLine f
       closeFile f
       return (line ++ line2)
   else
       return line
```

Rewriting the functions `openFile`, `readLine` and `closeFile` in a similar manner to above yields the following types:

```
openFile≀ :: FilePath ≀→ IO_L 1 File
openFile≀ = openFile ≀

readLine≀ :: File ≀⊸ IO_L 1 (File, Unrestricted Bytestring)
readLine≀ = readLine ≀

closeFile≀ :: File ≀⊸ IO_L ω ()
closeFile≀ = closeFile ≀
```

We reuse our definition of $;^{\wr}$, and define a special assignment operator $<-^{\wr}$ similarly to `let`$^{\wr}$, as follows:

$$x <-^{\wr} e \stackrel{\text{def}}{=} \wr, x <- e$$

With these components we can rewrite the above to the following:

```
do openFile≀ "myFile.txt" ;≀
   line <-≀ readLine≀
   if somePredicate line then
       line2 <-≀ readLine≀
       closeFile≀ ;≀
       return (line ++ line2)
   else
       return line
```

2.2. Session type classes

Type classes [10, 21] provide type-safe ad-hoc polymorphism by means of constraints on parametrically polymorphic types. They allow the programmer to define a fixed set of functions over multiple datatypes, where each datatype has a bespoke implementation of each function in the set. We call these sets of functions type classes. They are usually implemented by *dictionary passing* [21]. That means that at compile time an additional argument (the dictionary) and suitable access to this argument are synthesised for all code depending on type classes. With implicit arguments we can make dictionary passing implicit, and type classes become a special case of implicit arguments. This is a common Scala idiom [15].

Implicit messages suggest a natural generalisation of type classes: pass access to dictionaries by implicit messages! We illustrate this idea with a simple example. In Haskell, `Show` is a type class that converts values to their string representation. We generalise this to IM: instead of a conversion function, IM has a conversion *server*. We show two example implementations `intShow` and `boolShow` (we omit the details of the former). Additional function servers can be written against this code over types that define a `Show` type class server.

```
type Show = ?a.?ˡ⟨?a.!String.end⟩ᵃ.!String.end

show :: ⟨Show⟩ᵃ → end
show c =
    let         c = accept c           in
    let a     , c = receive c          in
    let aShow , c = implicit receive c in
    let         d = request aShow      in
    let         d = send a d           in
    let as    , d = receive d          in
    send as c

implicit boolShow :: ⟨?Bool.!String.end⟩ᵃ → end
boolShow c =
    let     c = accept c  in
    let b , c = receive c in
    send (if b then "true" else "false") c

implicit intShow :: ⟨?Int.!String.end⟩ᵃ → end
intShow = ...

showUser :: ⟨Show⟩ʳ → end
showUser ap =
    let    c = request ap in
    let    c = send 10 c  in
    let s, c = receive c  in
    printf(s) ;
    c
```

Clients communicating with the `show` server such as `showUser` do not need explicitly to send their `show` implementation, but send one implicitly.

It would be possible to make this example even more terse by eliminating repeated rebinding with implicit functions, however for clarity we exhibit just one application of implicits at a time.

2.3. Context and dependency injection

Implicit functions are commonly used in Scala to pass contextual information to a large set of methods. If many methods require the same contextual information, explicitly passing this context to each method call as a parameter becomes laborious, and eliding this contextual information becomes desirable. It is a common Scala idiom to use implicit functions to elide this repetitious context passing.

This pattern can now easily be generalised to concurrency by eliding contextual information passing in client/server interaction. The following example is

an implementation of a simple web server, that receives a request from a client and dispatches each kind of request to an appropriate handler. Some contextual information is passed to each handler at the handler's dispatch time - this information might typically be a network configuration or database reference. An implementation without implicit messages requires the context to be passed to each handler each time one is spawned. This creates syntactic noise that the programmer might prefer to elide. It also introduces repetition which can lead to programmer error. Implicit messages allow us to omit this context passing by passing the context as an implicit message, decreasing repetition and thereby reducing the cognitive burden on the programmer.

Each handler eventually sends a response to the manager, with the result of its computation. The result can also be passed implicitly, further reducing the syntactic noise.

We show an implementation without implicit messages on the left, and an implementation with implicit messages on the right.

```
type Manager = &{
    service1: H̄₁
    ...
    serviceN: H̄ₙ
}
manager :: ⟨Manager⟩ → Ctx → end
manager c ctx = case c of {
    service1:
        let d = request handler1 in
        let d = send ctx d in
        ...
        let res, d = receive d in
        manager ctx
    ...
    serviceN:
        let d = request handlerN in
        let d = send ctx d in
        ...
        let res, d = receive d in
        manager ctx
}
handler1Impl :: H₁
handler1Impl =
    let d = accept handler1 in
    let ctx, d = receive d in
    ...
    let res = ... in
    let d = send res d in
...
handlerNImpl :: Hₙ
handlerNImpl =
    let d = accept handlerN in
    let ctx, d = receive d in
    ...
    let res = ... in
    let d = send res d in
```

```
type Manager = &{
    service1: H̄₁
    ...
    serviceN: H̄ₙ
}
manager :: ⟨Manager⟩ → Ctx ↦ end
manager c = case c of {
    service1:
        let d = request handler1 in
        ...
        let res, d = implicit receive d in
        manager

    ...
    serviceN:
        let d = request handlerN in
        ...
        let res, d = implicit receive d in
        manager

}
handler1Impl :: H₁
handler1Impl =
    let d = accept handler1 in
    let ≀, d = implicit receive d in
    ...
    let ≀ = ... in d

...
handlerNImpl :: Hₙ
handlerNImpl =
    let d = accept handlerN in
    let ≀, d = implicit receive d in
    ...
    let ≀ = ... in d
```

3. The language Im

This section presents the syntax of our language Im of implicit message passing. Im is a superset of Last. Last is a medium through which the idea of implicit message passing can be expressed. Its integration of functions and processes enables us to provide both: implicit functions and implicit messages.

As the compiler synthesises the missing arguments at compile-time from type information, calculi for implicit arguments might be best understood not as programming languages, but as meta-programming systems that generate code in a base language L from input programs in L with implicits. Indeed, SI [14], an extension of System F, Scala's foundations for implicits, does not have a self-contained operational semantics, and is instead compiled to System F. We use the same approach, and translate Im to Last.

3.1. Syntax

In the presentation of Im's syntax, let v range over values and e over expressions. We assume that x ranges over a countable set of term variables, c over a countable set of channel endpoints, n over $\mathbb{N} \cup \{\infty\}$, l over labels and I over finite subsets of $\mathbb{N}$. In order to make the presentation easily accessible, we highlight the extensions Im adds to Last.

$$
\begin{array}{rcl}
v & ::= & \lambda x.e \mid (v, v) \mid \texttt{unit} \mid \texttt{fix} \mid \texttt{fork} \\
& \mid & \texttt{request}\ n \mid \texttt{accept}\ n \mid \texttt{send} \\
& \mid & \texttt{receive} \mid \texttt{implicit receive} \\
\\
e & ::= & v \mid e\ e \mid (e, e) \mid \texttt{let}\ x, x = e\ \texttt{in}\ e \\
& \mid & \texttt{select}\ l\ e \mid \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} \\
& \mid & \wr \mid \texttt{let}\ x, \wr = e\ \texttt{in}\ e
\end{array}
$$

Here `implicit receive` is the implicit analog of `receive`. Unlike `receive`, it is not matched by a corresponding `send`, but a corresponding `send` is inserted during translation, while `implicit receive` is translated into a normal `receive`. $\wr$ denotes a query to the implicit scope. $\wr$ is removed at translation time, and is replaced by a nondeterministically chosen name in the implicit scope. This nondeterminism can be resolved via relatively simple heuristics, some of which are discussed in section 5.1. The construct `let x, ≀ = ...` allows us to add variables to the implicit scope, and as with the lone $\wr$, we also replace $\wr$ within `let` by a variable name during translation. Note that we often write `let ≀ = e in e'`. This is a convenience and can be thought of as syntactic sugar for `let _, ≀ = (_, e) in e'` where _ is an unused variable or expression.

The parameter n following `accept` n and `request` n gives a *bound* for session communication. This will be explained in later sections. Note that we omit the bound parameter for brevity where not relevant.

An Im program is a configuration of expressions in parallel, running as separate threads and typed in a suitable environment. We now define configurations,

ranged over by C.

$$\begin{array}{lcl} b & ::= & v \mid l \\ C & ::= & C \parallel C \mid c \mapsto (c, n, \vec{b}) \mid (\nu cc)C \mid \langle e \rangle \end{array}$$

4. Types for Im

Just as SI is given meaning by type-guided translation to System F in [14], we give such a translation of Im into LAST. This section prepares the translation by extending LAST's typing system with types for implicit message passing and implicit functions. Types for Im are given by the following grammar. Here T ranges over types for the λ-calculus part of Im, S over session types, and B over buffer types.

$$\begin{array}{lcl} T & ::= & \texttt{Unit} \mid S \mid T \otimes T \mid T \to T \mid T \multimap T \\ & \mid & \langle S \rangle^{\mathrm{r}} \mid \langle S \rangle^{\mathrm{a}} \mid \langle S, S' \rangle \mid T \wr\!\!\to T \\ & \mid & T \wr\!\!\multimap T \end{array}$$

$$\begin{array}{lcl} S & ::= & \texttt{end} \mid ?T.S \mid !T.S \mid \&\langle l_i : S_i \rangle_{i \in I} \\ & \mid & \oplus\langle l_i : S_i \rangle_{i \in I} \mid X \mid \mu X.S \mid ?^{\wr}T.S \\ & \mid & !^{\wr}T.S \\ B & ::= & T \mid l \end{array}$$

The type $T \wr\!\!\to T$ is the type of implicit functions. It is written $?\!\to$ in [14] but we replace $?$ by $\wr$ to avoid confusion with the input session type $?T.S$. The type $T \wr\!\!\multimap T$ is the linear equivalent of $T \wr\!\!\to T$. As with [14], we do not have syntax for implicit abstraction and application - these are inferred during *implicit resolution* in Section 5.

The types $!^{\wr}T.S$ and $?^{\wr}T.S$ are the types of implicit message input and output respectively. They are the dual of one another as with explicit output and input. Implicit output types cannot be deduced from a process's syntax (since they are implicit) and must be inferred by inspecting the process that contains the corresponding implicit input. This happens during implicit resolution.

Buffer content types $\vec{B}$ are composed of vectors of entries B. Each entry is either a type T, representing the type of a value that is to be sent and stored in the buffer, or a label l representing the selection of such an option l by a process communicating using the buffer. Buffer content types $\vec{B}$ are assigned to buffers $\vec{b}$ such that for each v in $\vec{b}$ there exists a type T in the corresponding buffer content type $\vec{B}$ such that $v : T$. This notion is made precise in Section 5.

4.0.1. Type schemas for constants

Given below are the type schemas for the constants k. They are the same as LAST's, and can be instantiated for any appropriate type.

$$\begin{array}{lcl}
\texttt{fix} & : & (T \to T) \to T \\
\texttt{send} & : & T \to !T.S \multimap S \\
\texttt{send} & : & T \to !T.S \to S \quad \text{if } un(T) \\
\texttt{fork} & : & T \to \texttt{Unit} \quad \text{if } un(T) \\
\texttt{receive} & : & ?T.S \to T \otimes S \\
\texttt{request } n & : & \langle S \rangle^{\mathrm{r}} \to \overline{S} \quad \text{if } bound(\overline{S}) \leq n \\
\texttt{accept } n & : & \langle S \rangle^{\mathrm{a}} \to S \quad \text{if } bound(S) \leq n \\
\texttt{unit} & : & \texttt{Unit}
\end{array}$$

Note that we omit a type schema for `implicit receive`. This is because it cannot be translated by the rule [T-Const] in Figure 1, but needs a bespoke typing rule as unlike the other constants its translation is not identity.

4.0.2. Session type duality

We now give the session type duality function for our calculus. If a session type S and S' are *dual*, written $\overline{S} = S'$, then a pair of terms of types S and S' can interact without communication errors. Such processes match in the sense that every action that one takes is matched by the other. If one outputs, the other inputs. If one offers a choice, the other makes a choice. We extend duality function of LAST to include the two forms of implicit communication:

$$\begin{array}{rclcrcl}
\overline{?T.S} & = & !T.\overline{S} & \qquad & \overline{!T.S} & = & ?T.\overline{S} \\
\overline{?^{\iota}T.S} & = & !^{\iota}T.\overline{S} & & \overline{!^{\iota}T.S} & = & ?^{\iota}T.\overline{S} \\
\overline{\mu X.S} & = & \mu X.\overline{S} & & \overline{X} & = & X \\
\overline{\oplus\langle l_i : S_i\rangle_{i\in I}} & = & \&\langle l_i : \overline{S}_i\rangle_{i\in I} & & \overline{\texttt{end}} & = & \texttt{end} \\
\overline{\&\langle l_i : S_i\rangle_{i\in I}} & = & \oplus\langle l_i : \overline{S}_i\rangle_{i\in I} & &
\end{array}$$

We now define the subtyping relation coinductively by extension of the definitions for LAST.

Definition 1. A type T is *contractive* if it does not have subexpressions of the form $\mu X_1 ... \mu X_n . X_i$ where $0 < i \leq n$.

Let $\mathscr{S}$ denote the set of contractive, closed session types, and let $\mathscr{T}$ denote the set of types in which all session types are contractive and closed. We now define

the function $F(\cdot)$ on binary relations over $\mathscr{T}$.

$$
\begin{aligned}
F(R) = \{&(\mathtt{end}, \mathtt{end})\} \\
&\cup \{(?T.S, ?T'.S') | (T, T'), (S, S') \in R\} \\
&\cup \{(!T.S, !T'.S') | (T', T), (S, S') \in R\} \\
&\cup \{(?^{\wr}T.S, ?^{\wr}T'.S') | (T, T'), (S, S') \in R\} \\
&\cup \{(!^{\wr}T.S, !^{\wr}T'.S') | (T', T), (S, S') \in R\} \\
&\cup \{(\&\langle l_i : S_i\rangle_{i\in I}, \&\langle l_j : S'_j\rangle_{j\in J}) \\
&\qquad | I \subseteq J, (S_i, S'_i) \in R, \forall i \in I\} \\
&\cup \{(\oplus\langle l_i : S_i\rangle_{i\in I}, \oplus\langle l_j : S'_j\rangle_{j\in J}) \\
&\qquad | J \subseteq I, (S_i, S'_i) \in R, \forall i \in J\} \\
&\cup \{(\langle S, S'\rangle, \langle S\rangle^{\mathrm{a}}) | S, S' \in \mathscr{S}\} \\
&\cup \{(\langle S, S'\rangle, \langle S'\rangle^{\mathrm{r}}) | S, S' \in \mathscr{S}\} \\
&\cup \{(\langle S\rangle^{\mathrm{a}}, \langle S'\rangle^{\mathrm{a}}) | (S, S') \in R\} \\
&\cup \{(\langle S\rangle^{\mathrm{r}}, \langle S'\rangle^{\mathrm{r}}) | (S, S') \in R\} \\
&\cup \{(\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle) | (S_1, S_2), (S'_1, S'_2) \in R\} \\
&\cup \{(T \to T', T \multimap T') | T, T' \in \mathscr{T}\} \\
&\cup \{(T_1 \to T'_1, T_2 \to T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
&\cup \{(T_1 \multimap T'_1, T_2 \multimap T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
&\cup \{(\mu X.S, S') | (S[^{\mu X.S}/_X], S') \in R\} \\
&\cup \{(S, \mu X.S') | (S, S'[^{\mu X.S'}/_X]) \in R\} \\
&\cup \{(T_1 \wr\!\!\to T'_1, T_2 \wr\!\!\to T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
&\cup \{(T_1 \wr\!\!\multimap T'_1, T_2 \wr\!\!\multimap T'_2) | (T_2, T_1), (T'_1, T'_2) \in R\} \\
&\cup \{(T \wr\!\!\to T', T \wr\!\!\multimap T') | T, T' \in \mathscr{T}\} \\
&\cup \{(T \to T', T \wr\!\!\to T') | T, T' \in \mathscr{T}\} \\
&\cup \{(T \multimap T', T \wr\!\!\multimap T') | T, T' \in \mathscr{T}\} \\
&\cup \{(!T.S, !^{\wr}T.S) | T \in \mathscr{T}, S \in \mathscr{S}\}
\end{aligned}
$$

Contractivity ensures that F is monotone. We write $T <: U$ if the pair (T, U) is in the greatest fixpoint of F. The last three lines in the definition of $F(\cdot)$ allow us to type the use of an explicit function in place of an implicit one, and the sending of explicit messages to implicit inputs. Such behaviour is allowed in Scala - the user may pass an explicit argument to an implicit function. We allow the same behaviour with implicit functions, and the analogous behaviour in the case of implicit messages.

The *matches* relation determines whether a given buffer type $\vec{B}$ agrees with a session type S. We write $\vec{B}\, mat\, S$ when the types in $\vec{B}$ match a prefix of those in S. We formalise this notion with the rules below:

$$\frac{\vec{B}\ \ mat\ \ S\quad U <: T}{U\vec{B}\ \ mat\ ?^{\iota}T.S}\ [\text{M-OutI}] \qquad \frac{\vec{B}\ \ mat\ \ S\quad U <: T}{U\vec{B}\ mat ?T.S}\ [\text{M-Out}]$$

$$\frac{}{\epsilon\ \ mat\ \ S}\ [\text{M-Empty}] \qquad \frac{\vec{B}\ \ mat\ \ S}{l\vec{B}\ \ mat\ \ \&\langle ..., l : S, ...\rangle}\ [\text{M-Case}]$$

For some S and $\vec{B}$ such that $\vec{B}\ \ mat\ \ S$, $S/\vec{B}$ gives the session behaviours remaining as a *postfix* of S after performing those behaviours that correspond with $\vec{B}$. We define the postfix operator below:

$$S/\epsilon = S \qquad ?T.S/U\vec{B} = S/\vec{B}$$
$$?^{\iota}T.S/U\vec{B} = S/\vec{B} \qquad \&\langle ..., l : S, ...\rangle/l\vec{B} = S/\vec{B}$$

4.0.3. Session type bounds

Next, we define $bound(S)$, which gives the *bound* of a session type, an upper bound on the runtime size of the buffer required to hold the values received on a channel with session type S. We start with the auxiliary operator $\mathsf{bds} \in (\mathscr{S} \to \mathbb{N}^{\infty}) \to \mathscr{S} \to \mathbb{N}^{\infty}$.

$$\mathsf{bds}(f)(S) = \begin{cases} 1 + f(S') & S \in \{?T.S', ?^{\iota}T.S'\} \\ 1 + max\{f(S_i)\}_{i \in I} & S = \&\langle l_i : S_i\rangle_{i \in I} \\ f(S[^{\mu X.S'}/_X]) & S = \mu X.S' \\ 0 & \text{otherwise} \end{cases}$$

We now define the relation $S \mapsto S'$, which computes an *advanced* session type S' given a session type S.

$$?T.S \mapsto S \qquad !T.S \mapsto S$$
$$?^{\iota}T.S \mapsto S \qquad !^{\iota}T.S \mapsto S$$
$$\&\langle ..., l : S, ...\rangle \mapsto S \qquad \oplus\langle ..., l : S, ...\rangle \mapsto S$$
$$\mu X.S \mapsto S' \text{ if } S\{\mu X.S/X\} \mapsto S'$$

We can now define $bound(S) = max\{\mu(S') | S \mapsto^* S'\}$ where μ is the least fixed point of bds.

5. Translation from Im to Last

This section presents implicit resolution, the type-directed translation of Im programs to Last. We proceed in three steps, translation of expressions, translation of buffers and translation of configurations. Following [14], the translation is type-directed in that we give typing rules for Im, instrumented with translations to Last. By forgetting the instrumentation, we obtain a typing system for Im.

$$\frac{\Gamma \vdash e : T \rightsquigarrow \widehat{e} \quad T <: U}{\Gamma \vdash e : U \rightsquigarrow \widehat{e}} \ [\text{T-Sub}] \qquad \frac{un(\Gamma) \quad k : T}{\Gamma \vdash k : T \rightsquigarrow k} \ [\text{T-Const}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2, x_1 : T, x_2 : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x_1, x_2 = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x_1, x_2 = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2} \ [\text{T-Split}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \rightsquigarrow \widehat{e}_1 \quad \Gamma_2 \vdash e_2 : U \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : T \otimes U \rightsquigarrow (\widehat{e}_1, \widehat{e}_2)} \ [\text{T-Pair}]$$

$$\frac{\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e} \quad un(\Gamma)}{\Gamma \vdash \lambda x.e : T \to U \rightsquigarrow \lambda x.\widehat{e}} \ [\text{T-Abs}] \qquad \frac{un(\Gamma)}{\Gamma, \alpha : T \vdash \alpha : T \rightsquigarrow \alpha} \ [\text{T-ID}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \multimap U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2 \vdash e_2 : T \rightsquigarrow \widehat{e}_2}{\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : U \rightsquigarrow \widehat{e}_1\ \widehat{e}_2} \ [\text{T-App}]$$

$$\frac{\Gamma_1 \vdash e : \&\langle l_i : T_i \rangle_{i \in I} \rightsquigarrow \widehat{e} \quad \forall_{i \in I}(\Gamma_2 \vdash e_i : T_i \multimap T \rightsquigarrow \widehat{e}_i)}{\Gamma_1 + \Gamma_2 \vdash \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} : T \rightsquigarrow \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e}_i\}_{i \in I}} \ [\text{T-Case}]$$

$$\frac{\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e}}{\Gamma \vdash \lambda x.e : T \multimap U \rightsquigarrow \lambda x.\widehat{e}} \ [\text{T-AbsL}] \qquad \frac{un(\Gamma)}{\Gamma, y : T \vdash \wr : T \rightsquigarrow y} \ [\text{T-Query}]$$

$$\frac{\Gamma \vdash e : \oplus\langle l_i : T_i \rangle_{i \in I} \rightsquigarrow \widehat{e} \quad j \in I}{\Gamma \vdash \texttt{select}\ l_j\ e : T_j \rightsquigarrow \texttt{select}\ l_j\ \widehat{e}} \ [\text{T-Select}]$$

$$\frac{\Gamma_1 \vdash e : T \wr\!\multimap U \rightsquigarrow \widehat{e} \quad \Gamma_2 \vdash \wr : T \rightsquigarrow y}{\Gamma_1 + \Gamma_2 \vdash e : U \rightsquigarrow \widehat{e}\ y} \ [\text{T-AppI}]$$

$$\frac{\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e} \quad y \text{ fresh} \quad un(\Gamma)}{\Gamma \vdash e : T \wr\!\to U \rightsquigarrow \lambda y.\widehat{e}} \ [\text{T-AbsI}]$$

$$\frac{\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1 \quad \Gamma_2, x : T, y : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2 \quad y \text{ fresh}}{\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, \wr = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2} \ [\text{T-SplitI}]$$

$$\frac{un(\Gamma)}{\Gamma \vdash \texttt{implicit receive} : ?^{\wr}T.S \to T \otimes S \rightsquigarrow \texttt{receive}} \ [\text{T-InI}]$$

$$\frac{\Gamma_1 \vdash \wr : T \rightsquigarrow y \quad \Gamma_2 \vdash e : !^{\wr}T.S \rightsquigarrow \widehat{e}}{\Gamma_1 + \Gamma_2 \vdash e : S \rightsquigarrow \texttt{send}\ y\ \widehat{e}} \ [\text{T-OutI}]$$

$$\frac{\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e} \quad y \text{ fresh}}{\Gamma \vdash e : T \wr\!\multimap U \rightsquigarrow \lambda y.\widehat{e}} \ [\text{T-AbsLI}]$$

Figure 1: Type guided translation of expressions.

5.0.1. Typing environments and implicit scope

Implicit resolution removes queries $\wr$ and inserts explicit functions and messages in place of implicit ones. This happens by choosing arguments from the implicit scope. We define the implicit scope thusly: The typing environment Γ is divided into two parts: the implicit and explicit scopes. That is to say, some of the bindings in Γ refer to implicit variables and some to explicit variables. In our typing rules we range over implicit variables with y and explicit variables with x. Variables enter the implicit scope in several ways: (1) when received as an implicit message; (2) when given as an argument to an implicit function; and (3) when bound by a `let` construct with $\wr$ on the left-hand side of the =.

5.0.2. Typing and translation of expressions

Typing judgements for expressions are of the form $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$. This can be read as: "under assumptions Γ, the IM expression e has type T and is translated to the LAST expression $\widehat{e}$". Our typing and translation rules can be found in Figure 1. With the exception of the new syntactic forms of expressions, the translations are homomorphic, yielding rules similar in structure to those found in [7]. The rules for our new syntactic forms are more interesting. The rules [T-SplitI], [T-AppI], [T-AbsI] and [T-Query] follow a similar structure to those in [14]. Note that with [T-Query], the variable chosen to replace $\wr$ must satisfy linearity constraints, a restriction not present in [14]. [T-AbsLI] is a linear version of the rule for implicit functions and is effectively a combination of the rules [T-AbsI] and [T-AbsL]. The rule [T-InI] translates `implicit receive` into `receive` and otherwise behaves in the same way as [T-Const]. [T-OutI] translates implicit outputs by inserting a `send` action into the process. The argument for the `send` is a variable from the implicit scope, which we get from the first premise by translating $\wr$ with (a subset of) the input environment. This yields an implicit variable with the appropriate type whilst also satisfying any linearity constraints. Note that [T-OutI] is the only rule adding outputs directly.

5.0.3. Typing and translation of buffer contents

Typing judgements for buffers follow the same form as typing judgements for expressions. We write $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$ in this case. The translation of buffers can be found in Figure 2.

5.0.4. Typing and translation of configurations

Typing judgements for configurations (Figure 3) follow a slightly different form to those for buffer contents and expressions. We write $\Gamma \vdash C \rhd \Delta \rightsquigarrow \widehat{C}$. This can be read as "under assumptions Γ, the configuration C yields buffer types Δ and is translated as $\widehat{C}$". We define buffer type maps Δ below in Definition 2. The rules [T-Thread], [T-Buffer] and [T-New] are as in [7], augmented with

$$\frac{\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}}{\Gamma \vdash l\vec{b} : l\vec{B} \rightsquigarrow \widehat{l\vec{b}}} \quad [\text{T-SeqL}] \qquad \frac{un(\Gamma)}{\Gamma \vdash \epsilon : \epsilon \rightsquigarrow \epsilon} \quad [\text{T-Empty}]$$

$$\frac{\Gamma_1 \vdash v : T \rightsquigarrow \widehat{v} \quad \Gamma_2 \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}}{\Gamma_1 + \Gamma_2 \vdash v\vec{b} : T\vec{B} \rightsquigarrow \widehat{v\vec{b}}} \quad [\text{T-SeqV}]$$

Figure 2: Type guided translation of buffers contents.

homomorphic translations. The rule [T-Par] [1] is also similar to its equivalent rule in [7], but also contains two new premises. The first computes the buffer types in the configuration $C_1 \parallel C_2$, which are used in the second premise to perform *implicit resolution*. The judgements used in these premises are explained below.

Definition 2. *Buffer types* are triples of the form $(d, n, \vec{B})$. We let Δ range over partial finite maps from channel names to *buffer types* in C. Writing $\Delta + \Delta'$ means that the domains of Δ and Δ' are disjoint.

Definition 3. We define a partial operation of addition on environments:

$$\Gamma + x : T = \begin{cases} \Gamma, x : T & x \notin \mathsf{dom}(\Gamma) \\ \Gamma & \Gamma(x) = T, \mathsf{un}(\Gamma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We extend this to $\Gamma + \Gamma'$ inductively from the base case.

5.1. Sources of ambiguity

There are two sources of ambiguity in implicit resolution. The first is in the selection of the implicit variable chosen by the rule [T-Query]. We do not specify which variable in the implicit scope should replace a $\wr$. A possible way to resolve this is to use nesting. Such a solution would select the innermost implicit variable of the appropriate type as the translation for $\wr$. The Scala compiler uses a more complex version of this strategy, augmented with other selection criteria [14].

The second source of ambiguity results from the insertion of output actions when resolving implicit messages. When a pair of composed processes are resolved, we do not specify which is resolved first. As a result, adjacent implicit inputs can be resolved in multiple ways. Consider the processes:

[1] Note that the rule [T-Par] of [7] uses a *compatibility* relation $S \asymp S'$, which holds exactly when $\overline{S} <: S'$. In this presentation we opt simply to write $\overline{S} <: S'$.

$$\frac{\Gamma \vdash e : T \rightsquigarrow \widehat{e} \quad un(T)}{\Gamma \vdash \langle e \rangle \rhd \emptyset \rightsquigarrow \langle \widehat{e} \rangle} \quad \text{[T-Thread]}$$

$$\frac{\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}} \quad |\vec{b}| \leq n}{\Gamma \vdash c \mapsto (d, n, \vec{b}) \rhd c : (d, n, \vec{B}) \rightsquigarrow c \mapsto (d, n, \widehat{\vec{b}})} \quad \text{[T-Buffer]}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma'_1 + \Gamma'_2 \quad \Gamma'_1 \vdash C_1 \rhd \Delta_1 \rightsquigarrow \widehat{C}_1 \quad \Gamma'_2 \vdash C_2 \rhd \Delta_2 \rightsquigarrow \widehat{C}_2 \quad \Delta' = \Delta_1 + \Delta_2 \\ \forall c \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B}) \Rightarrow (\vec{B} \ \ mat \ \ \Gamma'(c) \ and \ bound(\Gamma'(c)) \leq n)) \\ \forall c, d \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B}) \ and \ \Delta'(d) = (c, n', \vec{B'}) \Rightarrow \Gamma'(c)/\vec{B} <: \overline{\Gamma'(d)/\vec{B'}})\end{array}}{\Gamma \vdash C_1 \parallel C_2 \rhd \Delta' \rightsquigarrow \widehat{C}_1 \parallel \widehat{C}_2} \quad \text{[T-Par]}$$

$$\frac{\Gamma + c_1 : S_1 + c_2 : S_2 \vdash C \rhd \Delta + c_1 : (c_2, n_1, \vec{B}_1) + c_2 : (c_1, n_2, \vec{B}_2) \rightsquigarrow \widehat{C}}{\Gamma \vdash (\nu c_1 c_2) C \rhd \Delta \rightsquigarrow (\nu c_1 c_2) \widehat{C}} \quad \text{[T-New]}$$

Figure 3: Type guided translation of configurations.

```
⟨ let p =
    let ϰ = ...
    let c = accept x in
    let n, ϰ = implicit receive in c
in p ⟩ ‖ ⟨ let q =
    let ϰ = ...
    let d = request x in
    let n, ϰ = implicit receive in d
in q ⟩
```

Implicit resolution should insert two output actions here, one in p and the other in q. If we resolve p before q, we obtain the processes:

```
⟨ let p =
    let y = ...
    let c = accept x in
    send y c
    let n, c = receive in c
in p ⟩ ‖ ⟨ let q =
    let y = ...
    let d = request x in
    let n, d = receive in d
    send y d
in q ⟩
```

We could also resolve q first and obtain the processes:

```
⟨ let p =
    let y = ...
    let c = accept x in
    let n, c = receive in c
    send y c
in p ⟩ ‖ ⟨ let q =
    let y = ...
    let d = request x in
```

```
    send y d
    let n, d = receive in d
in q ⟩
```

As with ambiguity caused by resolution of $\wr$, an implementation could use a simple heuristic such as to resolve the left hand side of parallel composition before the right. In the absence of a proof, it is unclear whether such a heuristic could work in all possible cases. We leave these as future work.

6. Runtime safety of Im

We prove safety of Im's translation into LAST: we show that if we can derive $\Gamma \vdash C \triangleright \Delta \rightsquigarrow \widehat{C}$ in Im, then $\widehat{C}$ can be typed suitably in LAST. In order to make this precise, we define a function $(\cdot)^*$, that translates Im's types to standard LAST types. This translation simply erases occurrences of $\wr$, yielding non-implicit analogues of implicit types.

Definition 4 (Translation of types)**.**

$$
\begin{array}{rclcrcl}
(T \wr\!\!\to T')^* & = & T^* \to T'^* & \qquad & \texttt{Unit}^* & = & \texttt{Unit} \\
(T \wr\!\!\multimap T')^* & = & T^* \multimap T'^* & & \texttt{end}^* & = & \texttt{end} \\
(T \to T')^* & = & T^* \to T'^* & & (?T.S)^* & = & ?T^*.S^* \\
(T \multimap T')^* & = & T^* \multimap T'^* & & (!T.S)^* & = & !T^*.S^* \\
\&\langle l_i : S_i\rangle_{i\in I}^* & = & \&\langle l_i : S_i^*\rangle_{i\in I} & & (\langle S\rangle^{\mathsf{r}})^* & = & \langle S^*\rangle^{\mathsf{r}} \\
\oplus\langle l_i : S_i\rangle_{i\in I}^* & = & \oplus\langle l_i : S_i^*\rangle_{i\in I} & & (\langle S\rangle^{\mathsf{a}})^* & = & \langle S^*\rangle^{\mathsf{a}} \\
\langle S, S'\rangle^* & = & \langle S^*, S'^*\rangle & & (?^{\wr}T.S)^* & = & ?T^*.S^* \\
(T \otimes T')^* & = & T^* \otimes T'^* & & (!^{\wr}T.S)^* & = & !T^*.S^* \\
(\mu X.S)^* & = & \mu X.S^* & & X^* & = & X
\end{array}
$$

We extend the definition of $(\cdot)^*$ to buffer types:

Definition 5 (Translation of buffer types and environments)**.**

$$
\begin{array}{rclcrcl}
\epsilon^* & = & \epsilon & \qquad & (T\vec{B})^* & = & T^*\vec{B}^* \\
(l\vec{B})^* & = & l\vec{B}^* & & (c, n, \vec{B})^* & = & (c, n, \vec{B}^*)
\end{array}
$$

This is lifted pointwise to environments (i.e. $(\Gamma, x : T)^* = \Gamma^*, x : T^*$).

We call a configuration *fully buffered* if whenever it contains $c \mapsto (c', n, \vec{b})$ then it also contains $c' \mapsto (c, n', \vec{b'})$. We recall the following theorem from [7], defining and proving LAST's runtime safety.

Theorem (Runtime safety of LAST)**.** Let $\Gamma \vdash_{\text{LAST}} C \triangleright \Delta$ be a fully buffered LAST configuration, and assume that $C \longrightarrow^* C'$. If C'' is a blocked thread in C', then one if the following applies:

- C'' is $\langle v \rangle$ or $\langle \texttt{send}\ v \rangle$ or $\langle \texttt{request}\ n\ v \rangle$ or $\langle \texttt{accept}\ n\ v \rangle$;
- C'' is $\langle E[\texttt{receive}\ c] \rangle$ and $c \mapsto (\_, \_, \epsilon) \in C'$;
- C'' is $\langle E[\texttt{case}\ c\ \texttt{of}\ \{l_i : e_i\}_{i \in I}] \rangle$ and $c \mapsto (\_, \_, \epsilon) \in C'$.

This result was established in [7]. We now state our main result.

Theorem 1 (Runtime safety of Im)**.** If $\Gamma \vdash C \rhd \Delta \rightsquigarrow \widehat{C}$ is a fully buffered Im configuration, then $\Gamma^* \vdash_{\text{LAST}} \widehat{C} \rhd \Delta^*$ is a runtime-safe LAST configuration.

Proof. Immediately for Theorem 4. □

We now proceed to prove supporting lemmas and theorems, and theorem 4 itself.

Lemma 1 (Preservation of membership with $(\cdot)^*$ on environments)**.** $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma^*)$

Proof. Trivially by induction on Definition 5. □

Lemma 2 (Distributivity of $(\cdot)^*$ over $+$ on environments)**.** If $\Gamma_1 + \Gamma_2$ is defined, then $\Gamma_1^* + \Gamma_2^* = (\Gamma_1 + \Gamma_2)^*$

Proof. By induction on the definition of $+$.
There are two cases where $\Gamma + \alpha : T$ is defined:

- if $\alpha \notin dom(\Gamma)$, then $\Gamma + \alpha : T = \Gamma, \alpha : T$
  - Starting with $(\Gamma, \alpha : T)^*$, by the Definition 5, we get $\Gamma^*, \alpha : T^*$
  - Starting with $\Gamma^* + (\alpha : T)^*$, by the Definition 5, we get $\Gamma^* + \alpha : T^*$
    - Then, by the definition of $+$, we get $\Gamma^*, \alpha : T^*$
- if $\alpha : T \in \Gamma$ and $un(\Gamma)$, then $\Gamma + \alpha : T = \Gamma$
  - Starting with $(\Gamma + \alpha : T)^*$, we get Γ^* by the definition of $+$.
  - Starting with $\Gamma^* + (\alpha : T)^*$,
    - By Definition 5, $\Gamma^* + \alpha : T^*$
    - Then by Lemma 1, Γ^*.

Addition is extended inductively to a partial binary operation on environments, and distributivity therefore holds by the induction hypothesis. □

Lemma 3 (Preservation of contractivity and closure of types under translation)**.** If $T \in \mathscr{T}$, then $T^* \in \mathscr{T}_{LAST}$. Equally, if $S \in \mathscr{S}$, then $S^* \in \mathscr{S}_{LAST}$.

Proof. From Definition 4 we see that none of the translations change the number or position of type constructors in a type, and therefore contractivity is preserved. We also see that the names in μ-binders and of type variables are not modified by translation and thus closure is preserved. These can be shown formally by a routine induction on T, S. □

Lemma 4 (Preservation of subtyping with $(\cdot)^*$ on types)**.** If $T <: U$, then $T^* <:_{LAST} U^*$

Proof. We define $\mathcal{R} = \{(T^*, T'^*) \mid (T, T') \in \nu F\}$ and show that $\mathcal{R}$ is a prefixpoint of F_{LAST}, the monotone function [7] uses to define $<:_{LAST}$. In other words, we show that $\mathcal{R} \subseteq F_{LAST}(\mathcal{R})$. We proceed by induction on $(T, T') \in \nu F$.

- **Case** $(\mathtt{end}, \mathtt{end})$
  - To show: $(\mathtt{end}^*, \mathtt{end}^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\mathtt{end}, \mathtt{end}) \in \nu F_{LAST}$
  - The goal follows immediately from the definition of F_{LAST}.
- **Case** $(?T.S, ?T'.S')$
  - To show: $((?T.S)^*, (?T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(?T^*.S^*, ?T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(!T.S, !T'.S')$
  - To show: $((!T.S)^*, (!T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(!T^*.S^*, !T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(?^\wr T.S, ?^\wr T'.S')$
  - To show: $((?^\wr T.S)^*, (?^\wr T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(?T^*.S^*, ?T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(!^\wr T.S, !^\wr T'.S')$
  - To show: $((!^\wr T.S)^*, (!^\wr T'.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(!T^*.S^*, !T'^*.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T, T'), (S, S') \in \nu F$
    - Then by the induction hypothesis, $(T^*, T'^*), (S^*, S'^*) \in \nu F_{LAST}$
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(\&\langle l_i : S_i\rangle_{i\in I}, \&\langle l_j : S'_j\rangle_{j\in J})$
  - To show: $((\&\langle l_i : S_i\rangle_{i\in I})^*, (\&\langle l_j : S'_j\rangle_{j\in J})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\&\langle l_i : S^*_i\rangle_{i\in I}, \&\langle l_j : S'^*_j\rangle_{j\in J}) \in \nu F_{LAST}$
  - By the definition of F, $I \subseteq J, (S_i, S'_i) \in \nu F, \forall i \in I$.
    - Then by the induction hypothesis, $(S^*_i, S'^*_i) \in \nu F_{LAST}, \forall i \in I$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(\oplus\langle l_i : S_i\rangle_{i\in I}, \oplus\langle l_j : S'_j\rangle_{j\in J})$
  - To show: $((\oplus\langle l_i : S_i\rangle_{i\in I})^*, (\oplus\langle l_j : S'_j\rangle_{j\in J})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\oplus\langle l_i : S^*_i\rangle_{i\in I}, \oplus\langle l_j : S'^*_j\rangle_{j\in J}) \in \nu F_{LAST}$
  - By the definition of F, $I \subseteq J, (S_i, S'_i) \in \nu F, \forall i \in I$.
    - Then by the induction hypothesis, $(S^*_i, S'^*_i) \in \nu F_{LAST}, \forall i \in I$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(\langle S, S'\rangle, \langle S\rangle^{\mathtt{a}})$
  - To show: $(\langle S, S'\rangle^*, (\langle S\rangle^{\mathtt{a}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^*, S'^*\rangle, \langle S^*\rangle^{\mathtt{a}}) \in \nu F_{LAST}$
  - By the definition of F, $S, S' \in \mathscr{S}$.
    - Then by Lemma 3, $S^*, S'^* \in \mathscr{S}_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(\langle S, S'\rangle, \langle S'\rangle^{\mathtt{r}})$
  - To show: $(\langle S, S'\rangle^*, (\langle S'\rangle^{\mathtt{r}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^*, S'^*\rangle, \langle S'^*\rangle^{\mathtt{r}}) \in \nu F_{LAST}$
  - By the definition of F, $S, S' \in \mathscr{S}$.
    - Then by Lemma 3, $S^*, S'^* \in \mathscr{S}_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(\langle S\rangle^{\mathtt{a}}, \langle S'\rangle^{\mathtt{a}})$
  - To show: $((\langle S\rangle^{\mathtt{a}})^*, (\langle S'\rangle^{\mathtt{a}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^*\rangle^{\mathtt{a}}, \langle S'^*\rangle^{\mathtt{a}}) \in \nu F_{LAST}$
  - By the definition of F, $(S, S') \in \nu F$.
    - Then by the induction hypothesis, $(S^*, S'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(\langle S\rangle^{\mathtt{r}}, \langle S'\rangle^{\mathtt{r}})$
  - To show: $((\langle S\rangle^{\mathtt{r}})^*, (\langle S'\rangle^{\mathtt{r}})^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S^*\rangle^{\mathtt{r}}, \langle S'^*\rangle^{\mathtt{r}}) \in \nu F_{LAST}$

- By the definition of F, $(S, S') \in \nu F$.
  - Then by the induction hypothesis, $(S^*, S'^*) \in \nu F_{LAST}$.
  - The goal then follows from the definition of F_{LAST}.

- **Case** $(\langle S_1, S_1' \rangle, \langle S_2, S_2' \rangle)$
  - To show: $(\langle S_1, S_1' \rangle^*, \langle S_2, S_2' \rangle^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\langle S_1^*, S_1'^* \rangle, \langle S_2^*, S_2'^* \rangle) \in \nu F_{LAST}$
  - By the definition of F, $(S_1, S_1'), (S_2, S_2') \in \nu F$.
    - Then by the induction hypothesis, $(S_1^*, S_1'^*), (S_2^*, S_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T \to T', T \multimap T')$
  - To show: $((T \to T')^*, (T \multimap T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \to T'^*, T^* \multimap T'^*) \in \nu F_{LAST}$
  - By the definition of F, $T, T' \in \mathscr{T}$.
    - Then by Lemma 3, $T^*, T'^* \in \mathscr{T}_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T_1 \to T_1', T_2 \to T_2')$
  - To show: $((T_1 \to T_1')^*, (T_2 \to T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \to T_1'^*, T_2^* \to T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(T_1 \multimap T_1', T_2 \multimap T_2')$
  - To show: $((T_1 \multimap T_1')^*, (T_2 \multimap T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \multimap T_1'^*, T_2^* \multimap T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(\mu X.S, S')$
  - To show: $((\mu X.S)^*, S'^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(\mu X.S^*, S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(S[\mu X.S/X], S') \in \nu F$.
    - Then by the induction hypothesis, $(S^*[\mu X.S^*/X], S'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.

- **Case** $(S, \mu X.S')$
  - To show: $(S^*, (\mu X.S')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(S^*, \mu X.S'^*) \in \nu F_{LAST}$
  - By the definition of F, $(S, S'[^{\mu X.S'}/_X]) \in \nu F$.
    - Then by the induction hypothesis, $(S^*, S'^*[^{\mu X.S'^*}/_X]) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(T_1 \wr\!\!\rightarrow T_1', T_2 \wr\!\!\rightarrow T_2')$
  - To show: $((T_1 \wr\!\!\rightarrow T_1')^*, (T_2 \wr\!\!\rightarrow T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \rightarrow T_1'^*, T_2^* \rightarrow T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(T_1 \wr\!\!\multimap T_1', T_2 \wr\!\!\multimap T_2')$
  - To show: $((T_1 \wr\!\!\multimap T_1')^*, (T_2 \wr\!\!\multimap T_2')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T_1^* \multimap T_1'^*, T_2^* \multimap T_2'^*) \in \nu F_{LAST}$
  - By the definition of F, $(T_1, T_1'), (T_2, T_2') \in \nu F$.
    - Then by the induction hypothesis, $(T_1^*, T_1'^*), (T_2^*, T_2'^*) \in \nu F_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(T \wr\!\!\rightarrow T', T \wr\!\!\multimap T')$
  - To show: $((T \wr\!\!\rightarrow T')^*, (T \wr\!\!\multimap T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \rightarrow T'^*, T^* \multimap T'^*) \in \nu F_{LAST}$
  - By the definition of F, $T, T' \in \mathscr{T}$.
    - Then by Lemma 3, $T^*, T'^* \in \mathscr{T}_{LAST}$.
    - The goal then follows from the definition of F_{LAST}.
- **Case** $(T \rightarrow T', T \wr\!\!\rightarrow T')$
  - To show: $((T \rightarrow T')^*, (T \wr\!\!\rightarrow T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \rightarrow T'^*, T^* \rightarrow T'^*) \in \nu F_{LAST}$
  - The goal holds immediately by reflexivity of subtyping in LAST [7].
- **Case** $(T \multimap T', T \wr\!\!\multimap T')$
  - To show: $((T \multimap T')^*, (T \wr\!\!\multimap T')^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(T^* \multimap T'^*, T^* \multimap T'^*) \in \nu F_{LAST}$
  - The goal holds immediately by reflexivity of subtyping in LAST [7].

- **Case** $(!T.S, !^{l}T.S)$
  - To show: $((!T.S)^*, (!^{l}T.S)^*) \in \nu F_{LAST}$
    - Or by Definition 4, $(!T^*.S^*, !T^*.S^*) \in \nu F_{LAST}$
  - The goal holds immediately by reflexivity of subtyping in LAST [7].

□

Lemma 5 (Preservation of linearity and nonlinearity with $(\cdot)^*$ on types)**.** Let $T' = T^*$.

- $un(T) \iff un(T')$.
- $lin(T) \iff lin(T')$.

Proof. Trivially by induction on Definition 4. All linear types translate to linear types, and all unlimited types translate to unlimited types. □

Lemma 6 (Preservation of linearity and nonlinearity with $(\cdot)^*$ on environments)**.** Let $\Gamma' = \Gamma^*$.

- $un(\Gamma) \iff un(\Gamma')$.
- $lin(\Gamma) \iff lin(\Gamma')$.

Proof. Trivially by induction on Definition 5 and Lemma 5. □

Lemma 7 (Type-preserving translation of type schemes for constants)**.** For each constant type scheme $k : T$, there is a type scheme in LAST of the form $k : T^*$.

Proof. Follows immediately from Definition 4. □

Theorem 2 (Type-preserving translation of expressions)**.** Let e be an expression with implicits, let Γ be an environment that may contain implicit types, and let T be a type that may contain implicit types. If $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$, then $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$.

Proof. By induction on $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$

- **Case** $\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : T \otimes U \rightsquigarrow (\widehat{e}_1, \widehat{e}_2)$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} (\widehat{e}_1, \widehat{e}_2) : (T \otimes U)^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} (\widehat{e}_1, \widehat{e}_2) : (T \otimes U)^*$
    - or by Definition 4, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} (\widehat{e}_1, \widehat{e}_2) : T^* \otimes U^*$
  - By inversion of **T-Pair**:
    - $\Gamma_1 \vdash e_1 : T \rightsquigarrow \widehat{e}_1$
    - $\Gamma_2 \vdash e_2 : U \rightsquigarrow \widehat{e}_2$
  - By the induction hypothesis:

  - $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : T^*$
  - $\Gamma_2^* \vdash_{\text{LAST}} \widehat{e}_2 : U^*$
- By **T-Pair**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} (\widehat{e}_1, \widehat{e}_2) : T^* \otimes U^*$

- **Case** $\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : U \rightsquigarrow \widehat{e}_1\ \widehat{e}_2$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{e}_1\ \widehat{e}_2 : U^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}_1\ \widehat{e}_2 : U^*$
  - By inversion of **T-App**,
    - $\Gamma_2 \vdash e_2 : T \rightsquigarrow \widehat{e}_2$
    - $\Gamma_1 \vdash e_1 : T \multimap U \rightsquigarrow \widehat{e}_1$
  - By the induction hypothesis:
    - $\Gamma_2^* \vdash_{\text{LAST}} \widehat{e}_2 : T^*$
    - $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : (T \multimap U)^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : T^* \multimap U^*$
  - By **T-App**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}_1\ \widehat{e}_2 : U^*$

- **Case** $\Gamma \vdash \lambda x.e : T \to U \rightsquigarrow \lambda x.\widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : (T \to U)^*$
    - or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \to U^*$
  - By inversion of **T-Abs**:
    - $un(\Gamma)$
      - and by Lemma 6, $un(\Gamma^*)$
    - $\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $(\Gamma, x : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
      - then by Definition 5, $\Gamma^*, x : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - By **T-Abs**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \to U^*$

- **Case** $\Gamma \vdash \lambda x.e : T \multimap U \rightsquigarrow \lambda x.\widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : (T \multimap U)^*$
    - or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \multimap U^*$
  - By inversion of **T-AbsL**:
    - $\Gamma, x : T \vdash e : U \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $(\Gamma, x : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
      - then by Definition 5, $\Gamma^*, x : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - By **T-AbsL**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda x.\widehat{e} : T^* \multimap U^*$

- **Case** $\Gamma, \alpha : T \vdash \alpha : T \rightsquigarrow \alpha$
  - To show: $(\Gamma, \alpha : T)^* \vdash_{\text{LAST}} \alpha : T^*$

    - or by Definition 5, $\Gamma^*, \alpha : T^* \vdash_{\text{LAST}} \alpha : T^*$
  - By inversion of **T-ID**, $un(\Gamma)$
    - and by Lemma 6, $un(\Gamma^*)$
  - By **T-ID**$_{GV}$, $\Gamma^*, \alpha : T^* \vdash_{\text{LAST}} \alpha : T^*$
- **Case** $\Gamma \vdash k : T \rightsquigarrow k$
  - To show: $\Gamma^* \vdash_{\text{LAST}} k : T^*$
  - By inversion of **T-Const**, we have:
    - $un(\Gamma)$
      - and by Lemma 6, $un(\Gamma^*)$
    - $k : T$
      - and by Lemma 7, $k : T^*$
  - By **T-ID**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} k : T^*$
- **Case** $\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, y = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$
    - Or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$
  - By inversion of **T-Split**, we have:
    - $\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1$
      - and by the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : (T \otimes U)^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : T^* \otimes U^*$
    - $\Gamma_2, x : T, y : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2$
      - and by the induction hypothesis, $(\Gamma_2, x : T, y : U)^* \vdash_{\text{LAST}} \widehat{e}_2 : V^*$
      - then by Definition 5, $\Gamma_2^*, x : T^*, y : U^* \vdash_{\text{LAST}} \widehat{e}_2 : V^*$
  - By **T-Split**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$
- **Case** $\Gamma \vdash \texttt{select}\ l_j\ e : T_j \rightsquigarrow \texttt{select}\ l_j\ \widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \texttt{select}\ l_j\ \widehat{e} : T_j^*$
  - By inversion of **T-Select**, we have:
    - $\Gamma \vdash e : \oplus\langle l_i : T_i\rangle_{i \in I} \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : \oplus\langle l_i : T_i\rangle_{i \in I}^*$
      - then by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : \oplus\langle l_i : T_i^*\rangle_{i \in I}$
    - $j \in I$
  - By **T-Select**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \texttt{select}\ l_j\ \widehat{e} : T_j^*$
- **Case** $\Gamma_1 + \Gamma_2 \vdash \texttt{case}\ e\ \texttt{of}\ \{l_i : e_i\}_{i \in I} : T \rightsquigarrow \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e}_i\}_{i \in I}$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e}_i\}_{i \in I} : T^*$

- Or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e}_i\}_{i \in I} : T^*$

- By inversion of **T-Case**, we have:
  - $\Gamma_1 \vdash e : \&\langle l_i : T_i \rangle_{i \in I} \rightsquigarrow \widehat{e}$
    - and by the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : \&\langle l_i : T_i \rangle_{i \in I}^*$
    - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : \&\langle l_i : T_i^* \rangle_{i \in I}$
  - $\forall i \in I.\ \ \Gamma_2 \vdash e_i : T_i \multimap T \rightsquigarrow \widehat{e}_i$
    - and by the induction hypothesis, $\forall i \in I.\ \ \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}_i : (T_i \multimap T)^*$
    - then by Definition 4, $\forall i \in I.\ \ \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}_i : T_i^* \multimap T^*$
  - By **T-Case**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{case}\ \widehat{e}\ \texttt{of}\ \{l_i : \widehat{e}_i\}_{i \in I} : T^*$

- **Case** $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$
  - By inversion of **T-Sub**, we have:
    - $\Gamma \vdash e : U \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : U^*$
    - $T <: U$
      - and by Lemma 4, $T^* <: U^*$
  - Then by **T-Sub**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$

- **Case** $\Gamma, y : T \vdash \wr : T \rightsquigarrow y$
  - To show: $(\Gamma, y : T)^* \vdash_{\text{LAST}} y : T^*$
    - or by Definition 5, $\Gamma^*, y : T^* \vdash_{\text{LAST}} y : T^*$
  - By inversion of **T-Query**, $un(\Gamma)$
    - and by Lemma 6, $un(\Gamma^*)$
  - By **T-ID**$_{GV}$, $\Gamma^*, y : T^* \vdash_{\text{LAST}} y : T^*$

- **Case** $\Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x, \wr = e_1\ \texttt{in}\ e_2 : V \rightsquigarrow \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$
    - Or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$
  - By inversion of **T-SplitI**, we have:
    - $y\ fresh$
    - $\Gamma_1 \vdash e_1 : T \otimes U \rightsquigarrow \widehat{e}_1$
      - and by the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : (T \otimes U)^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e}_1 : T^* \otimes U^*$
    - $\Gamma_2, x : T, y : U \vdash e_2 : V \rightsquigarrow \widehat{e}_2$
      - and by the induction hypothesis, $(\Gamma_2, x : T, y : U)^* \vdash_{\text{LAST}} \widehat{e}_2 : V^*$
      - then by Definition 5, $\Gamma_2^*, x : T^*, y : U^* \vdash_{\text{LAST}} \widehat{e}_2 : V^*$

- By **T-Split**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \texttt{let}\ x, y = \widehat{e}_1\ \texttt{in}\ \widehat{e}_2 : V^*$

- **Case** $\Gamma_1 + \Gamma_2 \vdash e : U \rightsquigarrow \widehat{e}\ y$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{e}\ y : U^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}\ y : U^*$
  - By inversion of **T-AppI**,
    - $\Gamma_1 \vdash e : T \multimap U \rightsquigarrow \widehat{e}$
      - By the induction hypothesis, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : (T \multimap U)^*$
      - then by Definition 4, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{e} : T^* \multimap U^*$
    - $\Gamma_2 \vdash \wr : T \rightsquigarrow y$
      - By the induction hypothesis, $\Gamma_2^* \vdash_{\text{LAST}} y : T^*$
  - By **T-App**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{e}\ y : U^*$

- **Case** $\Gamma \vdash e : T \multimap U \rightsquigarrow \lambda y.\widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : (T \multimap U)^*$
    - or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \to U^*$
  - By inversion of **T-AbsI**:
    - $y\ fresh$
    - $un(\Gamma)$
      - and by Lemma 6, $un(\Gamma^*)$
    - $\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $(\Gamma, y : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
      - then by Definition 5, $\Gamma^*, y : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - By **T-Abs**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \to U^*$

- **Case** $\Gamma \vdash e : T \multimap U \rightsquigarrow \lambda y.\widehat{e}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : (T \multimap U)^*$
    - or by Definition 4, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \multimap U^*$
  - By inversion of **T-AbsI**:
    - $y\ fresh$
    - $\Gamma, y : T \vdash e : U \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $(\Gamma, y : T)^* \vdash_{\text{LAST}} \widehat{e} : U^*$
      - then by Definition 5, $\Gamma^*, y : T^* \vdash_{\text{LAST}} \widehat{e} : U^*$
  - By **T-AbsL**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \lambda y.\widehat{e} : T^* \multimap U^*$

- **Case** $\Gamma \vdash \texttt{implicit receive} : ?^{\wr}T.S \to T \otimes S \rightsquigarrow \texttt{receive}$
  - To show: $\Gamma^* \vdash \texttt{receive} : (?^{\wr}T.S \to T \otimes S)^*$
    - or, by Definition 4, $\Gamma^* \vdash \texttt{receive} : ?T^*.S^* \to T^* \otimes S^*$

- By inversion of **T-InI**, $un(\Gamma)$
  - and by Lemma 6, $un(\Gamma^*)$
- Result follows immediately from the type schema for `receive` and $\textbf{T-Const}_{GV}$.

- **Case** $\Gamma_1 + \Gamma_2 \vdash e : S \rightsquigarrow \texttt{send}\ y\ \widehat{e}$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$
  - By inversion of **T-OutI**,
    - $\Gamma_1 \vdash \wr : T \rightsquigarrow y$
      - and by the induction hypothesis, $\Gamma_1^* \vdash y : T^*$
    - $\Gamma_2 \vdash e : !^{\wr}T.S \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $\Gamma_2^* \vdash \widehat{e} : (!^{\wr}T.S)^*$
      - Then by Definition 4, $\Gamma_2^* \vdash \widehat{e} : !T^*.S^*$
  - **Case** $un(T)$:
    - By $\textbf{T-App}_{GV}$:
      - $\Gamma_1^* + \vdash \texttt{send}\ y : !T^*.S^* \to S^*$
      - and by Definition 5, $\Gamma_1^* \vdash \texttt{send}\ y : !T^*.S^* \to S^*$
    - Again $\textbf{T-App}_{GV}$:
      - $\Gamma_1^* + \Gamma_2^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$
      - Then by Lemma 2, $(\Gamma_1 + \Gamma_2)^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$
  - **Case** *not* $un(T)$:
    - By $\textbf{T-App}_{GV}$:
      - $\Gamma_1^* + \vdash \texttt{send}\ y : !T^*.S^* \multimap S^*$
      - and by Definition 5, $\Gamma_1^* \vdash \texttt{send}\ y : !T^*.S^* \multimap S^*$
    - Again $\textbf{T-App}_{GV}$:
      - $\Gamma_1^* + \Gamma_2^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$
      - Then by Lemma 2, $(\Gamma_1 + \Gamma_2)^* \vdash \texttt{send}\ y\ \widehat{e} : S^*$

□

Theorem 3 (Type and size-preserving translation of buffer contents)**.** Let $\vec{b}$ be a buffer with implicits, let Γ be an environment that may contain implicit types, and let $\vec{B}$ be a type vector that may contain implicit types. If $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$, then $\Gamma^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$.

Proof. By induction on $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$

- **Case** $\Gamma \vdash \epsilon : \epsilon \rightsquigarrow \epsilon$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \epsilon : \epsilon^*$
    - or by Definition 5, $\Gamma^* \vdash_{\text{LAST}} \epsilon : \epsilon$

- By inversion of **T-Empty**, $un(\Gamma)$
  - and by Lemma 5, $un(\Gamma^*)$
- By **T-Empty**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \epsilon : \epsilon$
- Trivially $|\epsilon| = |\epsilon|$

- **Case** $\Gamma_1 + \Gamma_2 \vdash v\vec{b} : T\vec{B} \rightsquigarrow \widehat{v}\widehat{\vec{b}}$
  - To show: $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : (T\vec{B})^*$
    - or by Definition 5, $(\Gamma_1 + \Gamma_2)^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : T^*\vec{B}^*$
    - or by Lemma 2, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : T^*\vec{B}^*$
  - By inversion of **T-SeqV**,
    - $\Gamma_1 \vdash v : T \rightsquigarrow \widehat{v}$
      - and by Theorem 2, $\Gamma_1^* \vdash_{\text{LAST}} \widehat{v} : T^*$
    - $\Gamma_2 \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$
      - and by the induction hypothesis, $\Gamma_2^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$
  - By **T-SeqV**$_{GV}$, $\Gamma_1^* + \Gamma_2^* \vdash_{\text{LAST}} \widehat{v}\widehat{\vec{b}} : T^*\vec{B}^*$
  - Since $|\vec{b}| = |\widehat{\vec{b}}|$, $|v| = 1$ and $|\widehat{v}| = 1$, then $|v\vec{b}| = |\widehat{v}\widehat{\vec{b}}|$

- **Case** $\Gamma \vdash l\vec{b} : l\vec{B} \rightsquigarrow \widehat{l\vec{b}}$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \widehat{l\vec{b}} : (l\vec{B})^*$
    - or by Definition 5, $\Gamma^* \vdash_{\text{LAST}} \widehat{l\vec{b}} : l\vec{B}^*$
  - By inversion of **T-SeqL**, $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$
    - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$
  - By **T-SeqL**$_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \widehat{l\vec{b}} : l\vec{B}^*$
  - Since $|\vec{b}| = |\widehat{\vec{b}}|$ and $|l| = 1$, then $|l\vec{b}| = |\widehat{l\vec{b}}|$

□

Lemma 8 (Preservation of matching under translation)**.** If $\vec{B}\ \ mat\ \ S$ then $\vec{B}^*\ \ mat\ \ S^*$

Proof. By induction on $\vec{B}\ \ mat\ \ S$.

- **Case** $\epsilon\ \ mat\ \ S$
  - To show: $\epsilon\ \ mat\ \ S^*$
  - Immediate from **M-Empty**

- **Case** $U\vec{B}\ mat?T.S$
  - To show: $(U\vec{B})^*\ \ mat\ \ (?T.S)^*$
    - or by Definition 5, $U^*\vec{B}^*\ \ mat\ \ (?T.S)^*$
    - or by Definition 4, $U^*\vec{B}^*\ mat\ ?T^*.S^*$
  - By inversion of **M-Out**,
    - $\vec{B}\ \ mat\ \ S$
      - and by the induction hypothesis, $\vec{B}^*\ \ mat\ \ S^*$
    - $U <: T$
      - and by Lemma 4, $U^* <: T^*$
  - By **M-Out**, $U^*\vec{B}^*\ mat\ ?T^*.S^*$
- **Case** $U\vec{B}\ mat?^{\backprime}T.S$
  - To show: $(U\vec{B})^*\ \ mat\ \ (?^{\backprime}T.S)^*$
    - or by Definition 5, $U^*\vec{B}^*\ \ mat\ \ (?^{\backprime}T.S)^*$
    - or by Definition 4, $U^*\vec{B}^*\ mat\ ?T^*.S^*$
  - By inversion of **M-Out**,
    - $\vec{B}\ \ mat\ \ S$
      - and by the induction hypothesis, $\vec{B}^*\ \ mat\ \ S^*$
    - $U <: T$
      - and by Lemma 4, $U^* <: T^*$
  - By **M-Out**, $U^*\vec{B}^*\ mat\ ?T^*.S^*$
- **Case** $l\vec{B}\ \ mat\ \ \&\langle ..., l : S, ...\rangle$
  - To show: $(l\vec{B})^*\ \ mat\ \ \&\langle ..., l : S, ...\rangle^*$
    - or by Definition 5, $l\vec{B}^*\ \ mat\ \ \&\langle ..., l : S, ...\rangle^*$
    - or by Definition 4, $l\vec{B}^*\ \ mat\ \ \&\langle ..., l : S^*, ...\rangle$
  - By inversion of **M-Case**, $\vec{B}\ \ mat\ \ S$
    - and by the induction hypothesis, $\vec{B}^*\ \ mat\ \ S^*$
  - By **M-Case**, $l\vec{B}^*\ \ mat\ \ \&\langle ..., l : S^*, ...\rangle$

□

Lemma 9 (Preservation of session type bounds under translation)**.** $bound(S) = bound(S^*)$

Proof. Immediate from the definitions of $bound(\cdot)$ and $bound_{LAST}(\cdot)$ in [7]. □

Lemma 10 (Preservation of duality under translation)**.** For all sessions S, $(\overline{S})^* = \overline{S^*}$.

Proof. By induction on S.

- **Case** $?T.S$
  - To show: $(\overline{?T.S})^* = \overline{(?T.S)^*}$
  - By Definition 4, $(\overline{?T.S})^* = \overline{?T^*.S^*}$
  - By the definition of duality, $(!T.\overline{S})^* = !T^*.\overline{S^*}$
  - By Definition 4, $!T^*.\overline{S}^* = !T^*.\overline{S^*}$
  - By the induction hypothesis, $!T^*.\overline{S^*} = !T^*.\overline{S^*}$
- **Case** $!T.S$
  - To show: $(\overline{!T.S})^* = \overline{(!T.S)^*}$
  - By Definition 4, $(\overline{!T.S})^* = \overline{!T^*.S^*}$
  - By the definition of duality, $(?T.\overline{S})^* = ?T^*.\overline{S^*}$
  - By Definition 4, $?T^*.\overline{S}^* = ?T^*.\overline{S^*}$
  - By the induction hypothesis, $?T^*.\overline{S^*} = ?T^*.\overline{S^*}$
- **Case** $?^\wr T.S$
  - To show: $(\overline{?^\wr T.S})^* = \overline{(?^\wr T.S)^*}$
  - By Definition 4, $(\overline{?^\wr T.S})^* = \overline{?T^*.S^*}$
  - By the definition of duality, $(!^\wr T.\overline{S})^* = !T^*.\overline{S^*}$
  - By Definition 4, $!T^*.\overline{S}^* = !T^*.\overline{S^*}$
  - By the induction hypothesis, $!T^*.\overline{S^*} = !T^*.\overline{S^*}$
- **Case** $!^\wr T.S$
  - To show: $(\overline{!^\wr T.S})^* = \overline{(!^\wr T.S)^*}$
  - By Definition 4, $(\overline{!^\wr T.S})^* = \overline{!T^*.S^*}$
  - By the definition of duality, $(?^\wr T.\overline{S})^* = ?T^*.\overline{S^*}$
  - By Definition 4, $?T^*.\overline{S}^* = ?T^*.\overline{S^*}$
  - By the induction hypothesis, $?T^*.\overline{S^*} = ?T^*.\overline{S^*}$
- **Case** $\oplus\langle l_i : S_i\rangle_{i\in I}$
  - To show: $\overline{\oplus\langle l_i : S_i\rangle_{i\in I}}^* = \overline{\oplus\langle l_i : S_i\rangle^*_{i\in I}}$
  - By Definition 4, $\overline{\oplus\langle l_i : S_i\rangle_{i\in I}}^* = \overline{\oplus\langle l_i : S_i^*\rangle_{i\in I}}$
  - By the definition of duality, $\&\langle l_i : \overline{S}_i\rangle^*_{i\in I} = \&\langle l_i : \overline{S^*}_i\rangle_{i\in I}$
  - By Definition 4, $\&\langle l_i : \overline{S}_i^*\rangle_{i\in I} = \&\langle l_i : \overline{S^*}_i\rangle_{i\in I}$
  - For all $i \in I$ by the induction hypothesis, $\&\langle l_i : \overline{S^*}_i\rangle_{i\in I} = \&\langle l_i : \overline{S^*}_i\rangle_{i\in I}$

- **Case** $\&\langle l_i : S_i \rangle_{i \in I}$
  - To show: $\overline{\&\langle l_i : S_i \rangle_{i \in I}}^* = \overline{\&\langle l_i : S_i \rangle^*_{i \in I}}$
  - By Definition 4, $\overline{\&\langle l_i : S_i \rangle_{i \in I}}^* = \overline{\&\langle l_i : S_i^* \rangle_{i \in I}}$
  - By the definition of duality, $\oplus\langle l_i : \overline{S}_i \rangle^*_{i \in I} = \oplus\langle l_i : \overline{S^*}_i \rangle_{i \in I}$
  - By Definition 4, $\oplus\langle l_i : \overline{S}_i^* \rangle_{i \in I} = \oplus\langle l_i : \overline{S^*}_i \rangle_{i \in I}$
  - For all $i \in I$ by the induction hypothesis, $\oplus\langle l_i : \overline{S^*}_i \rangle_{i \in I} = \oplus\langle l_i : \overline{S^*}_i \rangle_{i \in I}$
- **Case** `end`
  - To show: $(\overline{\texttt{end}})^* = \overline{\texttt{end}^*}$
  - By Definition 4, $(\overline{\texttt{end}})^* = \overline{\texttt{end}}$
  - By the definition of duality, $\texttt{end}^* = \texttt{end}$
  - By Definition 4, $\texttt{end} = \texttt{end}$
- **Case** X
  - To show: $\overline{X}^* = \overline{X^*}$
  - By Definition 4, $\overline{X}^* = \overline{X}$
  - By the definition of duality, $X^* = X$
  - By Definition 4, $X = X$
- **Case** $\mu X.S$
  - To show: $(\overline{\mu X.S})^* = \overline{(\mu X.S)^*}$
  - By Definition 4, $(\overline{\mu X.S})^* = \overline{\mu X.S^*}$
  - By the definition of duality, $(\mu X.\overline{S})^* = \mu X.\overline{S^*}$
  - By Definition 4, $\mu X.\overline{S}^* = \mu X.\overline{S^*}$
  - By the induction hypothesis, $\mu X.\overline{S^*} = \mu X.\overline{S^*}$

□

Lemma 11 (Preservation of postfix under translation)**.** For all sessions and buffers S, $\vec{B}$, $(S/\vec{B})^* = S^*/\vec{B}^*$

Proof. By induction on S, $\vec{B}$.

- **Case** S, ϵ:
  - To show: $(S/\epsilon)^* = S^*/\epsilon^*$
  - By the definition of postfixes, $S^* = S^*/\epsilon^*$
  - By Definition 5, $S^* = S^*/\epsilon$
  - By the definition of postfixes, $S^* = S^*$

- **Case** $?T.S,\ U\vec{B}$:
  - To show: $(?T.S/U\vec{B})^* = (?T.S)^*/(U\vec{B})^*$
  - By the definition of postfixes, $(S/\vec{B})^* = (?T.S)^*/(U\vec{B})^*$
  - By Definition 5, $(S/\vec{B})^* = (?T.S)^*/U^*\vec{B}^*$
  - By Definition 4, $(S/\vec{B})^* = ?T^*.S^*/U^*\vec{B}^*$
  - By the definition of postfixes, $(S/\vec{B})^* = S^*/\vec{B}^*$
  - By the induction hypothesis, $(S/\vec{B})^* = (S/\vec{B})^*$
- **Case** $?^\iota T.S,\ U\vec{B}$:
  - To show: $(?^\iota T.S/U\vec{B})^* = (?^\iota T.S)^*/(U\vec{B})^*$
  - By the definition of postfixes, $(S/\vec{B})^* = (?^\iota T.S)^*/(U\vec{B})^*$
  - By Definition 5, $(S/\vec{B})^* = (?^\iota T.S)^*/U^*\vec{B}^*$
  - By Definition 4, $(S/\vec{B})^* = ?T^*.S^*/U^*\vec{B}^*$
  - By the definition of postfixes, $(S/\vec{B})^* = S^*/\vec{B}^*$
  - By the induction hypothesis, $(S/\vec{B})^* = (S/\vec{B})^*$
- **Case** $\&\langle ..., l : S, ...\rangle,\ l\vec{B}$:
  - To show: $(\&\langle ..., l : S, ...\rangle/l\vec{B})^* = (\&\langle ..., l : S, ...\rangle)^*/(l\vec{B})^*$
  - By the definition of postfixes, $(S/\vec{B})^* = (\&\langle ..., l : S, ...\rangle)^*/(l\vec{B})^*$
  - By Definition 5, $(S/\vec{B})^* = (\&\langle ..., l : S, ...\rangle)^*/l\vec{B}^*$
  - By Definition 4, $(S/\vec{B})^* = \&\langle ..., l : S^*, ...\rangle/l\vec{B}^*$
  - By the definition of postfixes, $(S/\vec{B})^* = S^*/\vec{B}^*$
  - By the induction hypothesis, $(S/\vec{B})^* = (S/\vec{B})^*$

□

Theorem 4 (Type-preserving translation of configurations)**.** Let C be an configuration with implicits, and let Γ and Δ be environments that may contain implicit types. If $\Gamma \vdash C \rhd \Delta \rightsquigarrow \widehat{C}$, then $\Gamma^* \vdash_{\text{LAST}} \widehat{C} \rhd \Delta^*$.

Proof. By induction on $\Gamma \vdash C \rhd \Delta \rightsquigarrow \widehat{C}$

- **Case** $\Gamma \vdash \langle e\rangle \rhd \emptyset \rightsquigarrow \langle \widehat{e}\rangle$
  - To show: $\Gamma^* \vdash_{\text{LAST}} \langle \widehat{e}\rangle \rhd {}^*$
  - By inversion of **T-Thread**,
    - $\Gamma \vdash e : T \rightsquigarrow \widehat{e}$
      - and by the induction hypothesis, $\Gamma^* \vdash_{\text{LAST}} \widehat{e} : T^*$
    - $un(T)$

– and by Lemma 5, $un(T^*)$

– By $\textbf{T-Thread}_{GV}$, $\Gamma^* \vdash_{\text{LAST}} \langle \widehat{e} \rangle \rhd {}^*$

- **Case** $\Gamma \vdash c \mapsto (d, n, \vec{b}) \rhd c : (d, n, \vec{B}) \rightsquigarrow c \mapsto (d, n, \widehat{\vec{b}})$

  – To show: $\Gamma^* \vdash_{\text{LAST}} c \mapsto (d, n, \widehat{\vec{b}}) \rhd (c : (d, n, \vec{B}))^*$

  – or by Definition 5, $\Gamma^* \vdash_{\text{LAST}} c \mapsto (d, n, \widehat{\vec{b}}) \rhd c : (d, n, \vec{B}^*)$

  – By inversion of **T-Buffer**, $\Gamma \vdash \vec{b} : \vec{B} \rightsquigarrow \widehat{\vec{b}}$ and $|\vec{b}| \leq n$

  – By Theorem 3, $\Gamma^* \vdash_{\text{LAST}} \widehat{\vec{b}} : \vec{B}^*$ and $|\vec{b}| = |\widehat{\vec{b}}|$

  – and since $|\vec{b}| \leq n$ and $|\vec{b}| = |\widehat{\vec{b}}|$, $|\widehat{\vec{b}}| \leq n$

  – By Theorem 2, $\Gamma^* \vdash \widehat{\vec{b}} : \vec{B}^*$

  – By $\textbf{T-Buffer}_{GV}$, $\Gamma^* \vdash_{\text{LAST}} c \mapsto (d, n, \widehat{\vec{b}}) \rhd c : (d, n, \vec{B}^*)$

- **Case** $\Gamma \vdash C_1 \parallel C_2 \rhd \Delta \rightsquigarrow \widehat{C}_1 \parallel \widehat{C}_2$

  – To show: $\Gamma^* \vdash_{\text{LAST}} \widehat{C}_1 \parallel \widehat{C}_2 \rhd \Delta^*$

  – By inversion of **T-Par**,

    – $\Gamma' = \Gamma'_1 + \Gamma'_2$

    – $\Delta' = \Delta_1 + \Delta_2$

    – $\Gamma'_1 \vdash C_1 \rhd \Delta_1 \rightsquigarrow \widehat{C}_1$

    – $\Gamma'_2 \vdash C_2 \rhd \Delta_2 \rightsquigarrow \widehat{C}_2$

    – $\forall c \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B}) \Rightarrow (\vec{B}\ mat\ \Gamma'(c)\ and\ bound(\Gamma'(c)) \leq n))$

    – $\forall c, d \in dom(\Gamma') \cap dom(\Delta').(\Delta'(c) = (d, n, \vec{B})\ and\ \Delta'(d) = (c, n', \vec{B}') \Rightarrow \Gamma'(c)/\vec{B} <: \overline{\Gamma'(d)/\vec{B}'})$

  – By the induction hypothesis,

    – $\Gamma'^*_1 \vdash_{\text{LAST}} \widehat{C}_1 \rhd \Delta^*_1$

    – $\Gamma'^*_2 \vdash_{\text{LAST}} \widehat{C}_2 \rhd \Delta^*_2$

  – By Lemma 2,

    – $\Gamma'^* = (\Gamma'_1 + \Gamma'_2)^* = \Gamma'^*_1 + \Gamma'^*_2$

  – Trivially we have $\Delta'^* = (\Delta'_1 + \Delta'_2)^* = \Delta'^*_1 + \Delta'^*_2$

  – By Definitions 5 and 4,

    – $c \in dom(\Gamma') \cap dom(\Delta')\ and\ \Delta'(c) = (d, n, \vec{B}) \Rightarrow$
    $c \in dom(\Gamma'^*) \cap dom(\Delta'^*)\ and\ \Delta'^*(c) = (d, n, \vec{B}^*)$

    – then by Lemmas 8 and 9,
    $\forall c \in dom(\Gamma'^*) \cap dom(\Delta'^*).(\Delta'^*(c) = (d, n, \vec{B}^*) \Rightarrow$
    $(\vec{B}^*\ \ mat\ \ \Gamma'(c)^*\ and\ bound(\Gamma'(c)^*) \leq n))$

– From the definitions.
  – $\forall c, d \in dom(\Gamma'^*) \cap dom(\Delta'^*)$.
    $(\Delta'^*(c) = (d, n, \vec{B}^*)$ *and*
    $\Delta'^*(d) = (c, n', \vec{B}'^*) \Rightarrow \Gamma'^*(c)/\vec{B}^* \asymp \Gamma'^*(d)/\vec{B}'^*)$
– From the definitions.

- **Case** $\Gamma \vdash (\nu c_1 c_2)C \rhd \Delta \rightsquigarrow (\nu c_1 c_2)\widehat{C}$
  – To show: $\Gamma^* \vdash_{\text{LAST}} (\nu c_1 c_2)\widehat{C} \rhd \Delta^*$
  – By inversion of **T-New**,
    $\Gamma + c_1 : S_1 + c_2 : S_2 \vdash C \rhd \Delta + c_1 : (c_2, n_1, \vec{B}_1) + c_2 : (c_1, n_2, \vec{B}_2) \rightsquigarrow \widehat{C}$
  – By the induction hypothesis,
    $(\Gamma + c_1 : S_1 + c_2 : S_2)^* \vdash \widehat{C} \rhd (\Delta + c_1 : (c_2, n_1, \vec{B}_1) + c_2 : (c_1, n_2, \vec{B}_2))^*$
  – By Definition 5,
    $\Gamma^* + c_1 : S_1^* + c_2 : S_2^* \vdash \widehat{C} \rhd \Delta^* + c_1 : (c_2, n_1, \vec{B}_1)^* + c_2 : (c_1, n_2, \vec{B}_2)^*$
  – By Definition 5,
    $\Gamma^* + c_1 : S_1^* + c_2 : S_2^* \vdash \widehat{C} \rhd \Delta^* + c_1 : (c_2, n_1, \vec{B}_1^*) + c_2 : (c_1, n_2, \vec{B}_2^*)$
  – By $\textbf{T-New}_{GV}$, $\Gamma^* \vdash_{\text{LAST}} (\nu c_1 c_2)\widehat{C} \rhd \Delta^*$

□

7. The language MpIm

In the following sections we introduce the calculus MpIm, a multiparty session-typed π-calculus with implicit messages. We call the language *MultiParty Implicit Messages* (MpIm). As with Im, we give meaning to MpIm programs by a type-directed translation to a base calculus. Our base calculus for MpIm is the multiparty session-typed π-calculus of [6], which is typed according to section 1.3. We term this base calculus Mpst. We make one minor simplification to Mpst in our usage of it as the base calculus here: we disallow multicast output. Disallowing multicast output allows us to make several inconsequential syntactic simplifications that aid in brevity. Since the resulting calculus with this simplification is a subset of the full calculus with multicast output, all the soundness results that we depend on still hold.

Multiparty session-typed π-calculus is a model of computation based purely on message passing, unlike Last which is based on the λ-calculus. As such, the following formulation includes only implicit messages, and omits implicit functions, which are not applicable in a language without λ abstractions.

This formulation of implicit messages in another setting demonstrates the broad applicability of implicit messages to message-passing forms of computation. The formulation in another setting also shows the robustness of the translation-based approach.

The example uses of implicit messages sketched in section 2 are applicable in MpIm as well as Im. MpIm allows for more flexibility in the use of implicit

messages – for example, a process can interleave communication with a type class server and a third participant, which is not possible in Im (without sacrificing deadlock freedom).

7.1. Syntax

The grammar of MpIm is given in Figure 4. Note that x, y in P can also be $\wr$. We extend Mpst with four new syntactic constructs. The first, implicit value reception, written $c?^{\wr}(\mathtt{p}, x).P$, can be read "on channel c, implicitly receive a value from participant $\mathtt{p}$, and bind it to the name x, then perform actions P". The second, implicit channel reception, written $c?^{\wr}((\mathtt{q}, x)).P$ is similar, except that a channel is received as opposed to a value. The third, implicit channel hiding, written $(\nu\wr)P$, creates a fresh channel whose scope is P, accessible via an implicit query $\wr$. Finally, to the grammar of expressions, we add the implicit query $\wr$, whose behaviour is the same as in the language Im. As previously, we highlight additions to Mpst.

P	::=	$c?^{\wr}(\mathtt{p}, x).P$	Implicit Value Reception
	|	$c?^{\wr}((\mathtt{q}, x)).P$	Implicit Channel Reception
	|	$(\nu\wr)P$	Implicit Channel Hiding
	|	$c?(\mathtt{p}, x).P$	Value Reception
	|	$c?((\mathtt{q}, x)).P$	Channel Reception
	|	$(\nu a)P$	Channel Hiding
	|	$c!\langle \mathtt{p}, x\rangle.P$	Value Sending
	|	$c!\langle\langle \mathtt{p}, c'\rangle\rangle.P$	Channel Sending
	|	$c \oplus \langle \mathtt{p}, l\rangle.P$	Selection
	|	$c\&(\mathtt{p}, \{l_i : P_i\}_{i\in I})$	Branching
	|	$P \mid Q$	Parallel composition
	|	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional
	|	$\overline{u}[\mathtt{p}](y).P$	Multicast request
	|	$u[\mathtt{p}](y).P$	Accept
	|	def D in P	Recursion
	|	$X\langle e, c\rangle$	Process call
	|	$\mathbf{0}$	Inaction
D	::=	$X(x, y) = P$	Declaration
e	::=	$x \mid y$	Variable
	|	$\mathtt{true} \mid \mathtt{false}$	Boolean expression
	|	$e\ \mathtt{and}\ e'$	
	|	$\mathtt{not}\ e$	
x, y	::=	$\wr$	Implicit variable
	|	a	Explicit variable

Figure 4: Grammar of MpIm

8. Types for MPIM

Figure 5 shows the grammar of types in MPIM. We introduce two new session types. These are the dual types of implicit input and output, written $?^{\wr}(\mathtt{p}, U).T$ and $!^{\wr}p.U.T$ respectively. We also introduce two new global types. The first (below on the left), implicit exchange, represents implicit input/output at the global level. The second new global type represents sender-nondeterministic implicit exchange (below on the right).

$$\mathtt{p} \to \mathtt{q} :^{\wr}\langle U \rangle G \qquad\qquad \wr \to \mathtt{q} :^{\wr}\langle U \rangle G$$

The sender participant number is omitted and is selected during implicit resolution from the participant numbers in the outermost global type $\neq q$, provided there is a candidate participant with a value of the appropriate type in its scope at the point of exchange.

S	::=	$\mathtt{bool} \mid ... \mid G$	Sorts
U	::=	$S \mid \mathcal{T}$	Exchange types
T	::=	$?^{\wr}(\mathtt{p}, U).T$	Implicit Input
	|	$!^{\wr}\langle \mathtt{p}, U \rangle.T$	Implicit Output
	|	$?(\mathtt{p}, U).T$	Explicit Input
	|	$!\langle \mathtt{p}, U \rangle.T$	Explicit Output
	|	$\oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I} \rangle$	Selection
	|	$\&(\mathtt{p}, \{l_i : T_i\}_{i \in I})$	Branching
	|	$\mu t.T \mid t$	Recursion
	|	$\mathtt{end}$	Inaction
G	::=	$\mathtt{p} \to \mathtt{q} :^{\wr}\langle U \rangle.G$	Implicit Exchange
	|	$\wr \to \mathtt{q} :^{\wr}\langle U \rangle.G$	Implicit Exchange (Unspecified sender)
	|	$\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$	Branching
	|	$\mu t.G \mid t$	Recursion
	|	$\mathtt{end}$	Inaction

Figure 5: Grammar of types

8.1. Duality

The *dual* of a session type S, written $\overline{S}$, is the session type that can safely interact with S. We define duality inductively on the syntax of types. The definition is given in Figure 6. We extend the definition of duality of MPST to include the new implicit exchange types.

$$\mathtt{end} \bowtie \mathtt{end} \qquad t \bowtie t \qquad \mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow \mu t.\mathfrak{T} \bowtie \mu t.\mathfrak{T}'$$

$$\mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow !U.\mathfrak{T} \bowtie ?U.\mathfrak{T}' \qquad \mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow ?U.\mathfrak{T} \bowtie !U.\mathfrak{T}'$$

$$\mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow !^{\wr}U.\mathfrak{T} \bowtie ?^{\wr}U.\mathfrak{T}' \qquad \mathfrak{T} \bowtie \mathfrak{T}' \Longrightarrow ?^{\wr}U.\mathfrak{T} \bowtie !^{\wr}U.\mathfrak{T}'$$

$$\forall i \in I.\mathfrak{T}_i \bowtie \mathfrak{T}'_i \Longrightarrow \oplus\{l_i : \mathfrak{T}_i\}_{i\in I} \bowtie \&\{l_i : \mathfrak{T}'_i\}_{i\in I}$$

$$\exists i \in I.l = l_i \wedge \mathfrak{T} \bowtie \mathfrak{T}_i \Longrightarrow \oplus l; \mathfrak{T} \bowtie \&\{l_i : \mathfrak{T}_i\}_{i\in I}$$

Figure 6: Type duality

8.2. Global Type Projection

Figure 7 shows the global projection of the generalised type G onto $\mathtt{q}$. Our global type projection differs from that of MPST in that we parameterise the projection by an additional participant number $\mathtt{m}$, which is intended as the maximum participant number of the 'outermost' global session type G, such that $mp(G) = \mathtt{m}$. Where our typing rules project a global type G onto a participant number $\mathtt{q}$, they write $G \upharpoonright_{mp(G)} \mathtt{q}$ (or equivalent), The maximum participant number is required by the function as an additional argument since sender-nondeterministic implicit inputs do not specify which participant is to provide it with an implicit value, but a participant number is chosen when implicits are resolved. The global projection function selects a participant number to be the sender that matches the implicit receive, but the projection function should not select a participant number $\mathtt{p}$ outside the range $1 \leq \mathtt{p} \leq mp(G)$. The participant number choices in the cases for sender-nondeterministic implicit exchanges are guaranteed to fall within this allowable range since they are passed the maximum participant number allowed and include checks that verify that it is not exceeded.

8.3. Partial Type Projection

Figure 8 shows the partial projection of the generalised type τ onto $\mathtt{q}$, denoted by $\tau \upharpoonright \mathtt{q}$. Our partial type projection more closely resembles its MPST counterpart than our global type projection – we simply extend it to our new session types $?^{\wr}(\mathtt{p}, U).T$ and $!^{\wr}p.U.T$.

9. Translation from MPIM to MPST

The typing and translation rules for MPIM expressions are given in Figure 9, and the rules for processes in Figure 10.

Our type system utilises two *binding rules* (IMBIND, EXBIND) which use a two place judgement of the form $x \rightsquigarrow a$. Since the binders in MPIM are allowed to be $\wr$, binding a received message to the implicit scope in the case of input,

$$(\mathtt{p} \to \mathtt{p}' : \langle U \rangle.G) \upharpoonright_{\mathtt{m}} \mathtt{q} = \begin{cases} !\langle \mathtt{p}', U \rangle.(G \upharpoonright_{\mathtt{m}} \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p} \\ ?(\mathtt{p}, U).(G \upharpoonright_{\mathtt{m}} \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}' \\ G \upharpoonright_{\mathtt{m}} \mathtt{q} & \text{otherwise} \end{cases}$$

$$(\mathtt{p} \to \mathtt{p}' : \{l_i : G_i\}_{i \in I}) \upharpoonright_{\mathtt{m}} \mathtt{q} = \begin{cases} \oplus\langle \mathtt{p}', \{l_i : (G_i \upharpoonright_{\mathtt{m}} \mathtt{q})\}_{i \in I} \rangle & \text{if } \mathtt{q} = \mathtt{p} \\ \&(\mathtt{p}, \{l_i : (G_i \upharpoonright_{\mathtt{m}} \mathtt{q})\}_{i \in I}) & \text{if } \mathtt{q} = \mathtt{p}' \\ G_{i_0} & \text{if } \mathtt{q} \neq \mathtt{p}, \mathtt{q} \neq \mathtt{p}', i_0 \in I \\ & \text{and } \forall i, j \in I. G_i \upharpoonright_{\mathtt{m}} \mathtt{q} = G_j \upharpoonright_{\mathtt{m}} \mathtt{q} \end{cases}$$

$$(\mu t.G) \upharpoonright_{\mathtt{m}} \mathtt{q} = \begin{cases} \mu t.(G \upharpoonright_{\mathtt{m}} \mathtt{q}) & \text{if } G \upharpoonright_{\mathtt{m}} \mathtt{q} \neq t \\ \mathtt{end} & \text{otherwise} \end{cases} \qquad t \upharpoonright_{\mathtt{m}} \mathtt{q} = t \qquad \mathtt{end} \upharpoonright_{\mathtt{m}} \mathtt{q} = \mathtt{end}$$

$$(\mathtt{p} \to \mathtt{p}' :^{\wr}\langle U \rangle.G) \upharpoonright_{\mathtt{m}} \mathtt{q} = \begin{cases} !^{\wr}p'.U.(G \upharpoonright_{\mathtt{m}} \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p} \\ ?^{\wr}(\mathtt{p}, U).(G \upharpoonright_{\mathtt{m}} \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}' \\ G \upharpoonright_{\mathtt{m}} \mathtt{q} & \text{otherwise} \end{cases}$$

$$(\wr \to \mathtt{p} :^{\wr}\langle U \rangle.G) \upharpoonright_{\mathtt{m}} \mathtt{q} = \begin{cases} !^{\wr}p.U.(G \upharpoonright_{\mathtt{m}} \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p}' \text{ where } 1 \leq \mathtt{p}' \leq \mathtt{m} \\ ?^{\wr}(\mathtt{p}', U).(G \upharpoonright_{\mathtt{m}} \mathtt{q}) & \text{if } \mathtt{q} = \mathtt{p} \neq \mathtt{p}' \text{ where } 1 \leq \mathtt{p}' \leq \mathtt{m} \\ G \upharpoonright_{\mathtt{m}} \mathtt{q} & \text{otherwise} \end{cases}$$

Figure 7: Global Type Projection

and binding a fresh channel to the implicit scope in the case of restriction, we must convert $\wr$ binders to standard names in the results of translation to Mpst. Our binding rules handle the two possible cases: (ExBind), when the binder is a normal name, leaves the binder unchanged. (ImBind), which handles $\wr$, gives us a fresh standard name to replace $\wr$ in the translation. We add the fresh name to the typing environment, which allows us to replace $\wr$ with standard names in the components of the syntactic construct we are typing. Including binding premises in our typing rules for syntactic constructs with binders allows us to avoid duplicating typing rules. We avoid having separate rules for each construct with a binder, one for standard binders and one for implicit binders $\wr$.

The rules (ImRcv) and (ImSrcv) type implicit value and channel input respectively. The premises $x \rightsquigarrow a$, $y \rightsquigarrow a$ replace implicit binders $\wr$ with fresh names where necessary.

The rule (ImName) functions similarly to the rule (T-Query) from [9], choosing a type-appropriate value to insert in place of an implicit query.

The rules (ImSend) and (ImDeleg) synthesise outputs of values and channels respectively, where they are guided to do so by the appropriate process

$$(!\langle \mathtt{p}, U\rangle.T) \upharpoonright \mathtt{q} = \begin{cases} !U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases} \qquad (?(\mathtt{p}, U).T) \upharpoonright \mathtt{q} = \begin{cases} ?U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases}$$

$$(\oplus\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle) \upharpoonright \mathtt{q} = \begin{cases} \oplus\{l_i : T_i \upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p} \\ T_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p} \text{ and } \forall i, j \in I.\ T_i \upharpoonright \mathtt{q} = T_j \upharpoonright \mathtt{q} \end{cases}$$

$$(\&(\mathtt{p}, \{l_i : T_i\}_{i\in I})) \upharpoonright \mathtt{q} = \begin{cases} \&\{l_i : T_i \upharpoonright \mathtt{q}\}_{i\in I} & \text{if } \mathtt{q} = \mathtt{p} \\ T_1 \upharpoonright \mathtt{q} & \text{if } \mathtt{q} \neq \mathtt{p} \text{ and } \forall i, j \in I.\ T_i \upharpoonright \mathtt{q} = T_j \upharpoonright \mathtt{q} \end{cases}$$

$$(\mu t.T) \upharpoonright \mathtt{q} = \begin{cases} \mu t.(T \upharpoonright \mathtt{q}) & \text{if } T \upharpoonright \mathtt{q} \neq t \\ \mathtt{end} & \text{otherwise} \end{cases} \qquad t \upharpoonright \mathtt{q} = t \qquad \mathtt{end} \upharpoonright \mathtt{q} = \mathtt{end}$$

$$(!^{\wr} p.U.T) \upharpoonright \mathtt{q} = \begin{cases} !^{\wr} U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{q} = \mathtt{p} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases} \qquad (?^{\wr}(\mathtt{p}, U).T) \upharpoonright \mathtt{q} = \begin{cases} ?^{\wr} U.T \upharpoonright \mathtt{q} & \text{if } \mathtt{p} = \mathtt{q} \\ T \upharpoonright \mathtt{q} & \text{otherwise} \end{cases}$$

Figure 8: Partial Type Projection

types. The premise $\Gamma \vdash \wr : S \rightsquigarrow y$ in (ImSend) chooses an implicit value of the appropriate type to be send. (ImDeleg) has no such premise and instead chooses an unconsumed channel c from the session environment for delegation.

The rules (MCast) and (MAcc) type session request and acceptance respectively, and are very similar to their counterpart rules in Mpst, with the exception of new premises of form $\forall \mathtt{q} \in G.\lceil G \rceil_{\mathtt{p}} \upharpoonright \mathtt{q} = \lceil G \upharpoonright_{\mathtt{p}} \mathtt{q} \rceil$. These premises ensure that the participant numbers chosen to be inserted into sender-nondeterministic outputs match those chosen by the translation of the types in G. It is a form of consistency chech on the choice of participant numbers.

The rule (NRes) handles channel restriction/creation. Again a binding premise replaces $\wr$ with standard names where necessary.

9.1. Translation of types

As with Im to Last, we include a function to translate from MpIm types to Mpst types. We use this function in §10 as part of our soundness theorem. Our translation for session types is similar to the translation for Im's session types, but we also extend the translation to global types. As with projection for global types, our translation for global types is nondeterministic in its choice of participant numbers to replace $\wr$ where participant numbers occur in Mpst. As with global projection, we parameterise global translation with a participant number $\mathtt{m}$, intended to be the maximum participant number for the outermost global type.

We also use the translation function in the typing rules (MCast, MAcc) to check that the nondeterministic choices made by the global projection function agree with those made by the global translation function.

$$\frac{}{x \rightsquigarrow x}\ (\textsc{ExBind}) \qquad \frac{x \text{ fresh}}{\wr \rightsquigarrow x}\ (\textsc{ImBind})$$

$$\frac{}{\Gamma, y : S \vdash \wr : S \rightsquigarrow y}\ (\textsc{ImName}) \qquad \frac{}{\Gamma, x : S \vdash x : S \rightsquigarrow x}\ (\textsc{ExName})$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool} \rightsquigarrow \texttt{true}}\ (\textsc{True}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool} \rightsquigarrow \texttt{false}}\ (\textsc{False})$$

$$\frac{\Gamma \vdash e : \texttt{bool} \rightsquigarrow \widehat{e} \qquad \Gamma \vdash e' : \texttt{bool} \rightsquigarrow \widehat{e}'}{\Gamma \vdash e \texttt{ and } e' : \texttt{bool} \rightsquigarrow \widehat{e} \texttt{ and } \widehat{e}'}\ (\textsc{And})$$

Figure 9: Typing and translation rules for expressions

Definition 6 (Translation of types)**.** We define the *translation* of an MpIm type to a standard MPST type, written $\lceil S \rceil$, below: We extend the definition of $\lceil \bullet \rceil$ pointwise to standard environments Γ and session environments Δ.

10. Runtime safety of MpIm

In demonstrating the runtime safety of MpIm, our approach is similar to that used in demonstrating the runtime safety of Im: We show that if we can derive $\Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P}$, then $\widehat{P}$ can be typed suitably in Mpst, according to the typing rules in [6]. Again we make this precise using a translation function $\lceil \cdot \rceil$, which translates MpIm's types to standard Mpst types, defined in section 9.1.

Theorem 5 (Type-preserving translation of expressions)**.** If $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$ then $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \lceil S \rceil$.

Proof. By induction on typing judgements for expressions $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$. We omit judgements for syntax present in standard MPST as these cases are homomorphic.

- Case $\Gamma, y : S \vdash \wr : S \rightsquigarrow y$
  - To show: $\lceil \Gamma, y : S \rceil \vdash_{\text{MPST}} y : \lceil S \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil, y : \lceil S \rceil \vdash_{\text{MPST}} y : \lceil S \rceil$
  - The goal follows immediately from $(\textsc{Name})_{\text{MPST}}$.
- Case $\Gamma, x : S \vdash x : S \rightsquigarrow x$
  - To show: $\lceil \Gamma, x : S \rceil \vdash_{\text{MPST}} x : \lceil S \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil, x : \lceil S \rceil \vdash_{\text{MPST}} x : \lceil S \rceil$
  - The goal follows immediately from $(\textsc{Name})_{\text{MPST}}$.
- Case $\Gamma \vdash \texttt{true} : \texttt{bool} \rightsquigarrow \texttt{true}$

$$\frac{\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P} \qquad x \rightsquigarrow a}{\Gamma \vdash c?^{\wr}(\mathtt{q}, x).P \triangleright \Delta, c : ?^{\wr}(\mathtt{q}, S).T \rightsquigarrow c?(\mathtt{q}, a).\widehat{P}} \ (\textsc{ImRcv}) \qquad \frac{\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P} \qquad y \rightsquigarrow a}{\Gamma \vdash c?^{\wr}((\mathtt{q}, y)).P \triangleright \Delta, c : ?^{\wr}(\mathtt{q}, \mathcal{T}).T \rightsquigarrow c?((\mathtt{q}, a)).\widehat{P}} \ (\textsc{ImSrcv})$$

$$\frac{\Gamma \vdash \wr : S \rightsquigarrow y \qquad \Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash P \triangleright \Delta, c : !^{\wr}\langle \mathtt{p}, S\rangle.T \rightsquigarrow c!\langle \mathtt{p}, y\rangle.\widehat{P}} \ (\textsc{ImSend}) \qquad \frac{\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash P \triangleright \Delta, c : !^{\wr}\langle \mathtt{p}, \mathcal{T}\rangle.T, i : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, i\rangle\rangle.\widehat{P}} \ (\textsc{ImDeleg})$$

$$\frac{\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P} \qquad x \rightsquigarrow a}{\Gamma \vdash c?(\mathtt{q}, x).P \triangleright \Delta, c : ?(\mathtt{q}, S).T \rightsquigarrow c?(\mathtt{q}, a).\widehat{P}} \ (\textsc{Rcv}) \qquad \frac{\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P} \qquad y \rightsquigarrow a}{\Gamma \vdash c?((\mathtt{q}, y)).P \triangleright \Delta, c : ?(\mathtt{q}, \mathcal{T}).T \rightsquigarrow c?((\mathtt{q}, a)).\widehat{P}} \ (\textsc{Srcv})$$

$$\frac{\Gamma \vdash e : S \rightsquigarrow \widehat{e} \qquad \Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash c!\langle \mathtt{p}, e\rangle.P \triangleright \Delta, c : !\langle \mathtt{p}, S\rangle.T \rightsquigarrow c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P}} \ (\textsc{Send}) \qquad \frac{\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}}{\Gamma \vdash c!\langle\langle \mathtt{p}, c'\rangle\rangle.P \triangleright \Delta, c : !\langle \mathtt{p}, \mathcal{T}\rangle.T, c' : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P}} \ (\textsc{Deleg})$$

$$\frac{\begin{array}{c}\Gamma \vdash u : G \rightsquigarrow u \qquad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright_{\mathtt{p}} \mathtt{p} \rightsquigarrow \widehat{P} \\ \mathtt{p} = mp(G) \qquad \forall \mathtt{q} \in G.\lceil G \rceil_{\mathtt{p}} \upharpoonright \mathtt{q} = \lceil G \upharpoonright_{\mathtt{p}} \mathtt{q} \rceil\end{array}}{\Gamma \vdash \overline{u}[\mathtt{p}](y).P \triangleright \Delta \rightsquigarrow \overline{u}[\mathtt{p}](y).\widehat{P}} \ (\textsc{MCast})$$

$$\frac{\begin{array}{c}\Gamma \vdash u : G \rightsquigarrow u \qquad \Gamma \vdash P \triangleright \Delta, y : G \upharpoonright_{\mathtt{m}} \mathtt{p} \rightsquigarrow \widehat{P} \\ \mathtt{m} = mp(G) \qquad \mathtt{p} < mp(G) \qquad \forall \mathtt{q} \in G.\lceil G \rceil_{\mathtt{m}} \upharpoonright \mathtt{q} = \lceil G \upharpoonright_{\mathtt{m}} \mathtt{q} \rceil\end{array}}{\Gamma \vdash u[\mathtt{p}](y).P \triangleright \Delta \rightsquigarrow u[\mathtt{p}](y).\widehat{P}} \ (\textsc{MAcc})$$

$$\frac{\Gamma, a : G \vdash P \triangleright \Delta \rightsquigarrow \widehat{P} \qquad x \rightsquigarrow a}{\Gamma \vdash (\nu x)P \triangleright \Delta \rightsquigarrow (\nu a)\widehat{P}} \ (\textsc{NRes}) \qquad \frac{\Delta \ \mathtt{end} \text{ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta \rightsquigarrow \mathbf{0}} \ (\textsc{Inact})$$

$$\frac{\Gamma \vdash e : \mathtt{bool} \rightsquigarrow \widehat{e} \qquad \Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P} \qquad \Gamma \vdash P' \triangleright \Delta \rightsquigarrow \widehat{P'}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \triangleright \Delta \rightsquigarrow \mathtt{if}\ \widehat{e}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{P'}} \ (\textsc{If})$$

$$\frac{\Gamma \vdash P \triangleright \Delta, c : T_j \rightsquigarrow \widehat{P} \qquad j \in I}{\Gamma \vdash c \oplus \langle \mathtt{p}, l_j\rangle.P \triangleright \Delta, c : \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I}\rangle \rightsquigarrow c \oplus \langle \mathtt{p}, l_j\rangle.\widehat{P}} \ (\textsc{Sel})$$

$$\frac{\forall i \in I. \quad \Gamma \vdash P_i \triangleright \Delta, c : T_i \rightsquigarrow \widehat{P}_i}{\Gamma \vdash c\&(\mathtt{p}, \{l_i : P_i\}_{i \in I}) \triangleright \Delta, c : \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) \rightsquigarrow c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i \in I})} \ (\textsc{Branch})$$

$$\frac{\Gamma \vdash e : S \rightsquigarrow \widehat{e} \qquad \Delta \ \mathtt{end} \text{ only}}{\Gamma, X : S\ T \vdash X\langle e, c\rangle \triangleright \Delta, c : T \rightsquigarrow X\langle \widehat{e}, c\rangle} \ (\textsc{Var})$$

$$\frac{\Gamma, X : S\ t, a : S \vdash P \triangleright b : T \rightsquigarrow \widehat{P} \qquad \Gamma, X : S\ \mu t.T \vdash Q \triangleright \Delta \rightsquigarrow \widehat{Q} \qquad x \rightsquigarrow a \qquad y \rightsquigarrow b}{\Gamma \vdash \mathrm{def}\ X(x, y) = P\ \mathrm{in}\ Q \triangleright \Delta \rightsquigarrow \mathrm{def}\ X(a, b) = \widehat{P}\ \mathrm{in}\ \widehat{Q}} \ (\textsc{Def})$$

Figure 10: Typing and translation rules for processes

$$\lceil S \rceil = \begin{cases} \lceil G \rceil_{mp(G)} & \text{if } S = G \\ \lceil T \rceil & \text{if } S = T \\ S & \text{otherwise} \end{cases}$$

$$\lceil T \rceil = \begin{cases} !p.\lceil U \rceil.\lceil S' \rceil & \text{if } T = !p.U.S' \text{ or } !^{\wr}p.U.S' \\ ?(\mathtt{p}, \lceil U \rceil).\lceil S' \rceil & \text{if } T = ?(\mathtt{p}, U).S' \text{ or } ?^{\wr}(\mathtt{p}, U).S' \\ \oplus\langle \mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I} \rangle & \text{if } T = \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I} \rangle \\ \&(\mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I}) & \text{if } T = \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) \\ \mu t.\lceil T' \rceil & \text{if } T = \mu t.T' \\ T & \text{otherwise} \end{cases}$$

$$\lceil G \rceil_{\mathtt{m}} = \begin{cases} \mathtt{p} \to \mathtt{q} : \langle \lceil U \rceil \rangle.\lceil G \rceil_{\mathtt{m}} & \text{if } G = \mathtt{p} \to \mathtt{q} : \langle U \rangle.G \text{ or } \mathtt{p} \to \mathtt{q} :^{\wr}\langle U \rangle.G \\ \mathtt{p} \to \mathtt{q} : \langle \lceil U \rceil \rangle.\lceil G \rceil_{\mathtt{m}} & \text{if } G = \wr \to \mathtt{q} : \langle U \rangle.G \text{ or } \wr \to \mathtt{q} :^{\wr}\langle U \rangle.G \text{ where } 1 \leq \mathtt{p} \leq \mathtt{m} \\ \mathtt{p} \to \mathtt{q} : \{l_i : (\lceil G \rceil_{\mathtt{m}})_i\}_{i \in I} & \text{if } G = \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \\ \mu t.\lceil G \rceil_{\mathtt{m}} & \text{if } G = \mu t.G \\ G & \text{otherwise} \end{cases}$$

Figure 11: Translation of MPIM types to standard MPST types

  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathtt{true} : \lceil \mathtt{bool} \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathtt{true} : \mathtt{bool}$
  - The goal follows immediately from $(\text{BOOL})_{\text{MPST}}$.

- Case $\Gamma \vdash \mathtt{false} : \mathtt{bool} \rightsquigarrow \mathtt{false}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathtt{false} : \lceil \mathtt{bool} \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathtt{false} : \mathtt{bool}$
  - The goal follows immediately from $(\text{BOOL})_{\text{MPST}}$.

- Case $\Gamma \vdash e \ \mathtt{and} \ e' : \mathtt{bool} \rightsquigarrow \widehat{e} \ \mathtt{and} \ \widehat{e}'$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} \ \mathtt{and} \ \widehat{e}' : \lceil \mathtt{bool} \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} \ \mathtt{and} \ \widehat{e}' : \mathtt{bool}$
  - By inversion of (AND):
    - $\Gamma \vdash e : \mathtt{bool} \rightsquigarrow \widehat{e}$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \lceil \mathtt{bool} \rceil$
      - By Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \mathtt{bool}$
    - $\Gamma \vdash e' : \mathtt{bool} \rightsquigarrow \widehat{e}'$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e}' : \lceil \mathtt{bool} \rceil$
      - By Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e}' : \mathtt{bool}$
  - The goal then follows from $(\text{AND})_{\text{MPST}}$.

□

Lemma 12 (Preservation of participant numbers)**.** If $\mathtt{p} = mp(G)$ then $\mathtt{p} = mp(\lceil G \rceil)$. If $\mathtt{p} < mp(G)$ then $\mathtt{p} < mp(\lceil G \rceil)$.

Proof. Directly from the definition of $\lceil G \rceil$. □

Theorem 6 (Type-preserving translation of pure processes)**.** If $\Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P}$ then $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta \rceil$.

Proof. By induction on typing judgements for pure processes $\Gamma \vdash P : \Delta \rightsquigarrow \widehat{P}$. We omit judgements for syntax present in standard MPST as these cases are homomorphic.

- Case $\Gamma \vdash c?^{\wr}(\mathtt{q}, x).P \triangleright \Delta, c :?^{\wr}(\mathtt{q}, S).T \rightsquigarrow c?(\mathtt{q}, a).\widehat{P}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c?(\mathtt{q}, x).\widehat{P} \triangleright \lceil \Delta, c :?^{\wr}(\mathtt{q}, S).T \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c?(\mathtt{q}, x).\widehat{P} \triangleright \lceil \Delta \rceil, c :?(\mathtt{q}, \lceil S \rceil).\lceil T \rceil$
  - By inversion of (ImRcv):
    - $\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
      - And by the induction hypothesis: $\lceil \Gamma, a : S \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta, c : T \rceil$
      - By Definition 6: $\lceil \Gamma \rceil, a : \lceil S \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta \rceil, c : \lceil T \rceil$
      - By $(\text{Rcv})_{\text{MPST}}$: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c?(\mathtt{q}, a).\widehat{P} \triangleright \lceil \Delta \rceil, c :?(\mathtt{q}, \lceil S \rceil).\lceil T \rceil$
    - $x \rightsquigarrow a$, and then $a \neq \wr$ and either:
      - $a = x$, and the goal holds by $(\text{Rcv})_{\text{MPST}}$ with $c?(\mathtt{q}, a).\widehat{P} = c?(\mathtt{q}, x).\widehat{P}$
      - a fresh, and the goal holds by $(\text{Rcv})_{\text{MPST}}$ with $c?(\mathtt{q}, a).\widehat{P} \equiv_{\alpha} c?(\mathtt{q}, x).\widehat{P}$
- Case $\Gamma \vdash c?^{\wr}((\mathtt{q}, y)).P \triangleright \Delta, c :?^{\wr}(\mathtt{q}, \mathcal{T}).T \rightsquigarrow c?((\mathtt{q}, a)).\widehat{P}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c?((\mathtt{q}, a)).\widehat{P} \triangleright \lceil \Delta, c :?^{\wr}(\mathtt{q}, \mathcal{T}).T \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c?((\mathtt{q}, a)).\widehat{P} \triangleright \lceil \Delta \rceil, c :?(\mathtt{q}, \lceil \mathcal{T} \rceil).\lceil T \rceil$
  - By inversion of (ImSrcv):
    - $\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta, c : T, y : \mathcal{T} \rceil$
      - By Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta \rceil, c : \lceil T \rceil, y : \lceil \mathcal{T} \rceil$
      - By $(\text{Srcv})_{\text{MPST}}$: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c?((\mathtt{q}, y)).\widehat{P} \triangleright \lceil \Delta \rceil, c :?(\mathtt{q}, \lceil \mathcal{T} \rceil).\lceil T \rceil$
    - $y \rightsquigarrow a$, and then $a \neq \wr$ and either...
      - $a = y$, and the goal holds by $(\text{Srcv})_{\text{MPST}}$ with $c?((\mathtt{q}, a)).\widehat{P} = c?((\mathtt{q}, y)).\widehat{P}$
      - a fresh, and the goal holds by $(\text{Srcv})_{\text{MPST}}$ with $c?((\mathtt{q}, a)).\widehat{P} \equiv_{\alpha} c?((\mathtt{q}, y)).\widehat{P}$

- Case $\Gamma \vdash P \triangleright \Delta, c :!^{\wr}\langle \mathtt{p}, S\rangle.T \rightsquigarrow c!\langle \mathtt{p}, y\rangle.\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathtt{p}, y\rangle.\widehat{P} \triangleright \lceil\Delta, c :!^{\wr}\langle \mathtt{p}, S\rangle.T\rceil$
  - Or by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathtt{p}, y\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!^{\wr}\langle \mathtt{p}, \lceil S\rceil\rangle.\lceil T\rceil$
  - By inversion of (ImSend):
    - $\Gamma \vdash \wr : S \rightsquigarrow y$
      - By Theorem 5: $\lceil\Gamma\rceil \vdash_{\text{MPST}} y : \lceil S\rceil$
    - $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
      - By Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $(\text{Send})_{\text{MPST}}$.
- Case $\Gamma \vdash P \triangleright \Delta, c :!^{\wr}\langle \mathtt{p}, \mathcal{T}\rangle.T, i : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, i\rangle\rangle.P$
  - To show $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, i\rangle\rangle.\widehat{P} \triangleright \lceil\Delta, c :!^{\wr}\langle \mathtt{p}, \mathcal{T}\rangle.T, i : \mathcal{T}\rceil$
    - Or by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, i\rangle\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!\langle \mathtt{p}, \lceil\mathcal{T}\rceil\rangle.\lceil T\rceil, i : \lceil\mathcal{T}\rceil$
  - By inversion of (ImDeleg): $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
    - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
    - By Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $(\text{Deleg})_{\text{MPST}}$.
- Case $\Gamma \vdash c?(\mathtt{q}, x).P \triangleright \Delta, c :?(\mathtt{q}, S).T \rightsquigarrow c?(\mathtt{q}, a).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?(\mathtt{q}, a).\widehat{P} \triangleright \lceil\Delta, c :?(\mathtt{q}, S).T\rceil$
    - Or by Definition 6:$\lceil\Gamma\rceil \vdash_{\text{MPST}} c?(\mathtt{q}, a).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathtt{q}, \lceil S\rceil).\lceil T\rceil$
  - By inversion of (Rcv):
    - $\Gamma, a : S \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
      - And by the induction hypothesis: $\lceil\Gamma, a : S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
      - And by Definition 6: $\lceil\Gamma\rceil, a : \lceil S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
    - $x \rightsquigarrow a$, and then $a \neq \wr$ and either...
      - $a = x$, and the goal holds by $(\text{Rcv})_{\text{MPST}}$ with $c?(\mathtt{q}, a).\widehat{P} = c?(\mathtt{q}, x).\widehat{P}$
      - a fresh, and the goal holds by $(\text{Rcv})_{\text{MPST}}$ with $c?(\mathtt{q}, a).\widehat{P} \equiv_{\alpha} c?(\mathtt{q}, x).\widehat{P}$
- Case $\Gamma \vdash c?((\mathtt{q}, y)).P \triangleright \Delta, c :?(\mathtt{q}, \mathcal{T}).T \rightsquigarrow c?((\mathtt{q}, a)).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?((\mathtt{q}, a)).\widehat{P} \triangleright \lceil\Delta, c :?(\mathtt{q}, \mathcal{T}).T\rceil$
    - Or by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c?((\mathtt{q}, a)).\widehat{P} \triangleright \lceil\Delta\rceil, c :?(\mathtt{q}, \lceil\mathcal{T}\rceil).\lceil T\rceil$

- By inversion of (Srcv):
  - $\Gamma \vdash P \triangleright \Delta, c : T, y : \mathcal{T} \rightsquigarrow \widehat{P}$
    - Then by the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T, y : \mathcal{T}\rceil$
    - Then by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil, y : \lceil\mathcal{T}\rceil$
  - $y \rightsquigarrow a$, and then $a \neq \wr$ and either...
    - $a = y$, and the goal holds by $(\text{Srcv})_{\text{MPST}}$ with $c?((\mathtt{q}, a)).\widehat{P} = c?((\mathtt{q}, y)).\widehat{P}$
    - a fresh, and the goal holds by $(\text{Srcv})_{\text{MPST}}$ with $c?((\mathtt{q}, a)).\widehat{P} \equiv_\alpha c?((\mathtt{q}, y)).\widehat{P}$

- Case $\Gamma \vdash c!\langle \mathtt{p}, e\rangle.P \triangleright \Delta, c :!\langle \mathtt{p}, S\rangle.T \rightsquigarrow c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P} \triangleright \lceil\Delta, c :!\langle \mathtt{p}, S\rangle.T\rceil$
    - Or by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle \mathtt{p}, \widehat{e}\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!\langle \mathtt{p}, \lceil S\rceil\rangle.\lceil T\rceil$
  - By inversion of (Send):
    - $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$
      - Then by Theorem 5: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e} : \lceil S\rceil$
    - $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
      - Then by the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
      - Then by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $(\text{Send})_{\text{MPST}}$.

- Case $\Gamma \vdash c!\langle\langle \mathtt{p}, c'\rangle\rangle.P \triangleright \Delta, c :!\langle \mathtt{p}, \mathcal{T}\rangle.T, c' : \mathcal{T} \rightsquigarrow c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P} \triangleright \lceil\Delta, c :!\langle \mathtt{p}, \mathcal{T}\rangle.T, c' : \mathcal{T}\rceil$
    - Or by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} c!\langle\langle \mathtt{p}, c'\rangle\rangle.\widehat{P} \triangleright \lceil\Delta\rceil, c :!\langle \mathtt{p}, \lceil\mathcal{T}\rceil\rangle.\lceil T\rceil, c' : \lceil\mathcal{T}\rceil$
  - By inversion on (Deleg): $\Gamma \vdash P \triangleright \Delta, c : T \rightsquigarrow \widehat{P}$
    - Then by the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta, c : T\rceil$
    - Then by Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - The goal then holds by $(\text{Deleg})_{\text{MPST}}$.

- Case $\Gamma \vdash \overline{u}[\mathtt{p}](y).P \triangleright \Delta \rightsquigarrow \overline{u}[\mathtt{p}](y).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \overline{u}[\mathtt{p}](y).\widehat{P} \triangleright \lceil\Delta\rceil$
  - By inversion of (MCast):
    - $\Gamma \vdash u : G \rightsquigarrow u$
      - By Theorem 5: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u : \lceil G\rceil$
      - By Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u : \lceil G\rceil_{\mathtt{p}}$

- $\Gamma \vdash P \rhd \Delta, y : G \upharpoonright_{\mathtt{p}} \mathtt{p} \rightsquigarrow \widehat{P}$
  - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta, y : G \upharpoonright_{\mathtt{p}} \mathtt{p}\rceil$
  - By Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil, y : \lceil G \upharpoonright_{\mathtt{p}} \mathtt{p}\rceil$
- $\mathtt{p} = mp(G)$
  - by Lemma 12: $\mathtt{p} = mp(\lceil G\rceil)$
- $\forall \mathtt{q} \in G. \lceil G\rceil_{\mathtt{p}} \upharpoonright \mathtt{p} = \lceil G \upharpoonright_{\mathtt{p}} \mathtt{p}\rceil$
  - Then: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil, y : \lceil G\rceil_{\mathtt{p}} \upharpoonright \mathtt{p}$
  - Equally: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil, y : \lceil G\rceil \upharpoonright \mathtt{p}$
- The goal then holds by $(\text{MCast})_{\text{MPST}}$.

- Case $\Gamma \vdash u[\mathtt{p}](y).P \rhd \Delta \rightsquigarrow u[\mathtt{p}](y).\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u[\mathtt{p}](y).\widehat{P} \rhd \lceil\Delta\rceil$
  - By inversion of (MCast):
    - $\Gamma \vdash u : G \rightsquigarrow u$
      - By Theorem 5: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u : \lceil G\rceil$
      - By Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} u : \lceil G\rceil_{\mathtt{m}}$
    - $\Gamma \vdash P \rhd \Delta, y : G \upharpoonright_{\mathtt{m}} \mathtt{p} \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta, y : G \upharpoonright_{\mathtt{m}} \mathtt{p}\rceil$
      - By Definition 6: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil, y : \lceil G \upharpoonright_{\mathtt{m}} \mathtt{p}\rceil$
    - $\mathtt{p} < mp(G)$
      - By Lemma 12: $\mathtt{p} < mp(\lceil G\rceil)$
    - $\mathtt{m} = mp(G)$
    - $\forall \mathtt{q} \in G. \lceil G\rceil_{\mathtt{m}} \upharpoonright \mathtt{p} = \lceil G \upharpoonright_{\mathtt{m}} \mathtt{p}\rceil$
      - Then: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil, y : \lceil G\rceil_{\mathtt{m}} \upharpoonright \mathtt{p}$
      - Equally: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil, y : \lceil G\rceil \upharpoonright \mathtt{p}$
  - The goal then holds by $(\text{MAcc})_{\text{MPST}}$.

- Case $\Gamma \vdash (\nu x)P \rhd \Delta \rightsquigarrow (\nu a)\widehat{P}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} (\nu a)\widehat{P} \rhd \lceil\Delta\rceil$
  - By inversion of (NRes):
    - $\Gamma, a : G \vdash P \rhd \Delta \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil\Gamma, a : G\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil$
      - By Definition 6: $\lceil\Gamma\rceil, a : \lceil G\rceil \vdash_{\text{MPST}} \widehat{P} \rhd \lceil\Delta\rceil$
    - $x \rightsquigarrow a$, and then $a \neq \wr$ and either...
      - $a = x$, and the goal holds by $(\text{NRes})_{\text{MPST}}$ with $(\nu a)\widehat{P} = (\nu x)\widehat{P}$
      - a fresh, and the goal holds by $(\text{NRes})_{\text{MPST}}$ with $(\nu a)\widehat{P} \equiv_{\alpha} (\nu x)\widehat{P}$

- Case $\Gamma \vdash c \oplus \langle \mathtt{p}, l_j \rangle.P \triangleright \Delta, c : \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I} \rangle \rightsquigarrow c \oplus \langle \mathtt{p}, l_j \rangle.\widehat{P}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c \oplus \langle \mathtt{p}, l_j \rangle.\widehat{P} \triangleright \lceil \Delta, c : \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i \in I} \rangle \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c \oplus \langle \mathtt{p}, l_j \rangle.\widehat{P} \triangleright \lceil \Delta \rceil, c : \oplus\langle \mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I} \rangle$
  - By inversion of (Sel):
    - $j \in I$
    - $\Gamma \vdash P \triangleright \Delta, c : T_j \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta, c : T_j \rceil$
      - By Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta \rceil, c : \lceil T_j \rceil$
  - The goal then holds by $(\text{Sel})_{\text{MPST}}$.
- Case $\Gamma \vdash c\&(\mathtt{p}, \{l_i : P_i\}_{i \in I}) \triangleright \Delta, c : \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) \rightsquigarrow c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i \in I})$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i \in I}) \triangleright \lceil \Delta, c : \&(\mathtt{p}, \{l_i : T_i\}_{i \in I}) \rceil$
    - Or by Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} c\&(\mathtt{p}, \{l_i : \widehat{P}_i\}_{i \in I}) \triangleright \lceil \Delta \rceil, c : \&(\mathtt{p}, \{l_i : \lceil T \rceil_i\}_{i \in I})$
  - By inversion of (Branch): $\forall i \in I. \quad \Gamma \vdash P_i \triangleright \Delta, c : T_i \rightsquigarrow \widehat{P}_i$
    - By the induction hypothesis: $\forall i \in I. \quad \lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P}_i \triangleright \lceil \Delta, c : T_i \rceil$
    - By Definition 6: $\forall i \in I. \quad \lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P}_i \triangleright \lceil \Delta \rceil, c : \lceil T_i \rceil$
  - The goal then follows from $(\text{Branch})_{\text{MPST}}$.
- Case $\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \triangleright \Delta \rightsquigarrow \mathtt{if}\ \widehat{e}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{P}'$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathtt{if}\ \widehat{e}\ \mathtt{then}\ \widehat{P}\ \mathtt{else}\ \widehat{P}' \triangleright \lceil \Delta \rceil$
  - By inversion of (If):
    - $\Gamma \vdash e : \mathtt{bool} \rightsquigarrow \widehat{e}$
      - By Theorem 5: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \lceil \mathtt{bool} \rceil$
      - By Definition 6: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{e} : \mathtt{bool}$
    - $\Gamma \vdash P \triangleright \Delta \rightsquigarrow \widehat{P}$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil \Delta \rceil$
    - $\Gamma \vdash P' \triangleright \Delta \rightsquigarrow \widehat{P}'$
      - By the induction hypothesis: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \widehat{P}' \triangleright \lceil \Delta \rceil$
  - The goal then holds by $(\text{If})_{\text{MPST}}$.
- Case $\Gamma \vdash \mathbf{0} \triangleright \Delta \rightsquigarrow \mathbf{0}$
  - To show: $\lceil \Gamma \rceil \vdash_{\text{MPST}} \mathbf{0} \triangleright \lceil \Delta \rceil$
  - By inversion of (Inact): Δ `end` only
    - By Definition 6: $\lceil \Delta \rceil$ `end` only
  - The goal then holds by $(\text{Inact})_{\text{MPST}}$.

- Case $\Gamma, X : S\ T \vdash X\langle e, c\rangle \triangleright \Delta, c : T \rightsquigarrow X\langle \widehat{e}, c\rangle$
  - To show: $\lceil \Gamma, X : S\ T\rceil \vdash_{\text{MPST}} X\langle \widehat{e}, c\rangle \triangleright \lceil \Delta, c : T\rceil$
    - Or by Definition 6: $\lceil\Gamma\rceil, X : \lceil S\rceil\ \lceil T\rceil \vdash_{\text{MPST}} X\langle \widehat{e}, c\rangle \triangleright \lceil\Delta\rceil, c : \lceil T\rceil$
  - By inversion of (Var):
    - $\Gamma \vdash e : S \rightsquigarrow \widehat{e}$
      - By Theorem 5, $\lceil\Gamma\rceil \vdash_{\text{MPST}} \widehat{e} : \lceil S\rceil$
    - Δ `end` only
      - By Definition 6, $\lceil\Delta\rceil$ `end` only
  - The goal then follows from $(\text{Var})_{\text{MPST}}$.
- Case $\Gamma \vdash \mathbf{def}\ X(x, y) = P\ \mathbf{in}\ Q \triangleright \Delta \rightsquigarrow \mathbf{def}\ X(a, b) = \widehat{P}\ \mathbf{in}\ \widehat{Q}$
  - To show: $\lceil\Gamma\rceil \vdash_{\text{MPST}} \mathbf{def}\ X(a, b) = \widehat{P}\ \mathbf{in}\ \widehat{Q} \triangleright \lceil\Delta\rceil$
  - By inversion of (Def):
    - $\Gamma, X : S\ t, a : S \vdash P \triangleright b : T \rightsquigarrow \widehat{P}$
      - And by the induction hypothesis: $\lceil\Gamma, X : S\ t, a : S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright \lceil b : T\rceil$
      - And by Definition 6: $\lceil\Gamma\rceil, X : \lceil S\rceil\ t, a : \lceil S\rceil \vdash_{\text{MPST}} \widehat{P} \triangleright b : \lceil T\rceil$
    - $\Gamma, X : S\ \mu t.T \vdash Q \triangleright \Delta \rightsquigarrow \widehat{Q}$
      - And by the induction hypothesis: $\lceil\Gamma, X : S\ \mu t.T\rceil \vdash_{\text{MPST}} \widehat{Q} \triangleright \lceil\Delta\rceil$
      - And by Definition 6: $\lceil\Gamma\rceil, X : \lceil S\rceil\ \mu t.\lceil T\rceil \vdash_{\text{MPST}} \widehat{Q} \triangleright \lceil\Delta\rceil$
    - $x \rightsquigarrow a$, and then $a \neq \wr$ and either $a = x$ or a fresh
    - $y \rightsquigarrow b$, and then $b \neq \wr$ and either $b = x$ or b fresh
  - The goal then holds by $(\text{Def})_{\text{MPST}}$ with $\mathbf{def}\ X(a, b) = \widehat{P}\ \mathbf{in}\ \widehat{Q}$ being either equal to, or α-equivalent to $\mathbf{def}\ X(x, y) = \widehat{P}\ \mathbf{in}\ \widehat{Q}$.

□

11. Further work

We have generalised the concept of implicit functions from Scala's sequential setting to message passing concurrency, established the soundness of the our proposal by translation, and demonstrated the usefulness of implicit message passing by examples from the literature, including a coherent solution to the repeated rebinding problem of linearly typed languages. Our approach to implicit messages generalises straightforwardly to multiparty communication, demonstrating the robustness of our approach.

Implicit choice of participant numbers is a feature clearly suggested to us by the formalism. We leave finding examples where this is useful in practice as future work.

Implicits are useful in sequential programming not just for type classes, but also for generic programming [16]. Our encoding of type classes as sessions leveraging implicits provides evidence that they provide an avenue for investigation into generic programming for message passing systems, as well as for sequential programming. Such a technology transfer from the domain of sequential to concurrent computation would be aided by a better understanding of the relationship between implicit functions and implicit messages.

Milner's groundbreaking work on functions as processes [13] gave a deep understanding of λ-calculi as processes engaging only in well-structured interaction. Can we use Milner's approach to clarify the exact correspondence between implicit functions and implicit messages? A precise match between SI and Im is unlikely, because the calculi are too different: e.g. SI's bidirectional and parametrically polymorphic typing system, which Im lacks. Clearly those choices are orthogonal to implicits, and we conjecture that full abstraction results between System F and binary session-typed π-calculus [4, 20] remain stable when source and target calculi are extended with implicits. In order even to be able to state full abstraction we need to generalise the existing equational theories (as well as reasoning tools like typed bisimulations) to λ- and π-calculi with implicits. The nature of any correspondence between a functional language like System F and multiparty session-typed π-calculus is an open question, and thus it is less clear whether such a correspondence could hold in the multiparty context when the source and target calculus are extended with implicits.

Another open issue is the resolution of ambiguity for implicit message passing. [14] discusses this problem in the context of System F, but modern Scala is based on dependent object types (DOT) [1, 2, 17]. It is interesting to extend DOT with implicit functions, and study the resolution of ambiguity arising from implicits using the approach to resolution implemented in Dotty, Scala's DOT-based compiler.

Finally, this paper advertised the utility of implicit messages, but as its argument had to rely on moderately-sized examples, and the aesthetic appeal of smooth generalisation from sequential to concurrent computation, a more substantial empirical evaluation is desirable. Unfortunately, the well-known difficulties with empirical evaluation of programming languages (see [11] for an overview) are aggravated here by the absence of mainstream programming language with session-typed message passing concurrency and implicit messages. Consequently we must leave robust empirical evaluation of implicit messages as future work.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council of the United Kingdom, and the University of Sussex. We thank D. Castro, S. Gay, A. Scalas, V. Vasconcelos and the anonymous reviewers of [9], for valuable feedback.

[1] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World*, pages 249–272. Springer, 2016.

[2] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, number EPFL-CONF-183030, 2012.

[3] R. Atkey. Parameterised notions of computation. *JFP*, 19(3-4):335–376, 2009.

[4] M. Berger, K. Honda, and N. Yoshida. Genericity and the π-calculus. *Acta Informatica*, 2005.

[5] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, Dec. 2017.

[6] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 2015.

[7] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 20(1):19âĂŞ50, 2010.

[8] K. Honda. Types for dyadic interaction. In *Proc. CONCUR*, 1993.

[9] A. Jeffery and M. Berger. Asynchronous sessions with implicit functions and messages. International Symposium on Theoretical Aspects of Software Engineering (TASE), 2018.

[10] S. Kaes. Parametric Overloading in Polymorphic Programming Languages. In *Proc. ESOP*, pages 131–144, 1988.

[11] A.-J. Kaijanaho. *Evidence-based programming language design: a philosophical and methodological exploration.* PhD thesis, University of Jyväskylä, 2015.

[12] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: Dynamic scoping with static types. In *Proc. POPL*, 2000.

[13] R. Milner. Functions as Processes. *MSCS*, 2(2):119–141, 1992.

[14] M. Odersky, O. Blanvillain, F. Liu, A. Biboudis, H. Miller, and S. Stucki. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. POPL*, 2018.

[15] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proc. OOPSLA*, pages 341–360, 2010.

[16] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: A new foundation for generic programming. In *Proc. PLDI*, pages 35–44, 2012.

[17] T. Rompf and N. Amin. Type Soundness for Dependent Object Types (DOT). In *ACM Sigplan Notices*, volume 51, pages 624–641. ACM, 2016.

[18] D. C. Sobral and M. Braun. Where does scala look for implicits? `https://web.archive.org/web/20181119203031/https://docs.scala-lang.org/tutorials/FAQ/finding-implicits.html`, 2011.

[19] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *Proc. PARLE*, 1994.

[20] B. Toninho and N. Yoshida. On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings. *CoRR*, abs/1711.00878, 2017.

[21] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proc. POPL*, pages 60–76, 1989.